Phil’s Technology Answers

Phil Jones

Phil’s Technology Answers

Phil Jones

Contents

The Programming Life

Intro

What are some things that programmers know, but most people don't?

That someone has to make the hard choices that turn informal requirements into formal, inflexible rules.

Most people, even people who like to think of themselves as doing useful work in tech. like designers / UX people / systems analysts / managers etc., can talk about what the computer should do in a way which leaves certain things vague and unspecified. People can communicate with each other while leaving certain details to the imagination of the listener.

The programmer is the person who doesn't have that luxury. He or she knows that the computer "deciding about X" needs to become a set of logical expressions and that everything has to fall definitively one side or the other when we test something. Everything will become black and white. There will never be comfortable grey areas. To mechanize means that everything must be commited to, one way or the other. And if you ignore that, you'll just end up with hard commitments that you don't even know you've made.

See : http://www.quora.com/Computer-Programmers/What-are-some-things-that-programmers-know-but-most-people-dont

Which careers have their roots in both philosophy and technology?

I always say that software development is "applied metaphysics". In metaphysics you ask how the world is really organised. In software development you try to figure out the most convenient way of modeling the world for your application. http://en.m.wikipedia.org/wiki/H… (http://en.m.wikipedia.org/wiki/Haecc
just the difference between identity and value in object orientation etc.

See : http://www.quora.com/Technology/Which-careers-have-their-roots-in-both-philosophy-and-technology

Can Computer Science be explained in terms of philosophy, like science sometimes is?

There's a lot of overlap between CS and philosophy.

Some computer science (not all) is a kind of maths. And maths shares with philosophy that it's an abstract field, studying ideas in principle, independent of their empirical consequences. So, some CS is like some philosophy. In fact some core knowledge (eg. logic) is fundamental to both philosophy AND CS. (Truth Tables were invented by Wittgenstein for example.)

I call computer science "applied metaphysics" in the sense that metaphysicians ask how the world really IS. (What it's fundamentally made of. Not as in "atoms" but in terms of really abstract things like "substances" or "essences" or "properties" or "haccaeties") Computer scientists ask how the world is best "modelled" for our practical purposes : objects or relations? values or mutable structures? types? etc.

I wouldn't agree that physics or other sciences are reducable to philosophy (as in your lowest energy-state example). There's no reason to think that the empirically observed behaviour of the universe that we capture with laws has ANYTHING to do with our philosophical notions. Or rather, philosophy may or may not help us to frame hypothesis. But it doesn't EXPLAIN them. Our scientific theories are not true BECAUSE they correspond to philsophical analogies. They are true because of how the universe really is.

So, yes, there's a lot of parallels between CS and philosophy. People who are good at, and like the kind of thinking involved in, one tend to have good intuitions about the other, even if they've never trained in it.

OTOH, science isn't really a relevant comparison. Science doesn't have that relation with philosophy so the CS connection with philosophy is not "like science" in that sense.

See : http://www.quora.com/Computer-Science/Can-Computer-Science-be-explained-in-terms-of-philosophy-like-science-sometimes-is

Is there a TV series about computer programmers?

?

Given how bad it could have been, The IT Crowd is amazingly good. But it still has to depend on the whole "geeks' problems relating to non-geeks" cliche for its dynamic.

The problem is that what's fascinating about the geek's world is extremely abstract. And television does not DO abstract. Television's strengths are intimacy and emotion, close-ups of facial expressions and body language. Series where you can get an deep understanding of character as you watch him or her evolve over many episodes (sometimes over years.)

But, frankly, programmers' "characters" aren't particularly interesting. Or even particularly varied. Sometimes you'll see a huge religious argument over language or text-editor or static vs dynamic typing, and when you look at the participants you discover that their characters are almost identical. They don't represent deep differences of human personality : the good vs. the bad, the rash vs. the responsible etc.

The joy of programming is that it's a world where *ideas themselves* matter most. And not because they merely illustrate personalities or power-struggles.

But television is a lousy medium for talking about ideas. Ideas are made of words, and words aren't visual. So TV's pictures are irrelevant at best and distractions and red-herrings at worst. Even TV's vocabulary is a distraction. Instead of talking about ideas, TV will want to use personality clashes as a *proxy* for differences of ideas. But what the viewer will take home is the personality differences, not the idea differences.

TV will want to invent dramatic arcs : overcoming difficulties, enlightening realizations, breaking or mending relationships. None of these things matter much in the world of ideas either. The best arguments don't become personal. Enlightenment happens through reading some good articles or Quora answers, not as cathartic moments of deep personal crises.

See : http://www.quora.com/Computer-Programming/Is-there-a-TV-series-about-computer-programmers

What are some cultural faux pas among programmers?

You know that job I gave you two weeks to write? Actually I need it for the meeting next week so I'm putting John to help you on it.

See : http://www.quora.com/Cultural-Faux-Pas/What-are-some-cultural-faux-pas-among-programmers

Are you really a geek or are you doing that to impress your geek girlfriend:o

If only …

See : http://www.quora.com/Are-you-really-a-geek-or-are-you-doing-that-to-impress-your-geek-girlfriend-o

Passion

Does being a programmer makes the one disconnected or less interested in life?

Being a programmer is all about learning to switch between different levels of abstraction when thinking about something. You have to be able to hold highly abstract algorithms in your head and then swoop down to think how they can be represented in the programming language, or even deeper to start figuring out what's in memory or on the stack etc. while debugging your code.

Understanding "abstraction" means moving from the particular to the general case and back again.

Once you get this habit, it's very easy to apply it to the rest of your life. You start to see much of what goes on around you as mere examples of patterns or systems. This *can* be a great comfort, as many things which would otherwise seem like personal slights and problems just start to look like mere instances of patterns. Why should I take this personally? That's just what X people do. Or how a Y situation tends to play out.

But inevitably this comfort and calmness is a kind of disengagement. You care less for the individuals, both people and things, in your life and more for the processes that bring them to you. But those individuals have their own value. Your girl or boy-friend is not just an instance of a class of RomanticPartner but someone who expects you to take their uniqueness and irreplaceability seriously. Your things should be things with sentimental value because of the particular history, not just their role in your life.

Become too comfortable with the world of abstractions and you may lose attachment to the particulars. You will care about things that seem to most people around you as TOO ABSTRACT, eg. poltiics, policy, economics, ecology while simultaneously losing track of the particulars that mean everything to them.

See : http://www.quora.com/Does-being-a-programmer-makes-the-one-disconnected-or-less-interested-in-life

Does working in engineering destroy and take the happiness out of your life?

No … working in engineering is awesome : it can be creatively fulfilling, economically lucrative AND socially useful. There are relatively few professions that tick all three boxes.

See : http://www.quora.com/Does-working-in-engineering-destroy-and-take-the-happiness-out-of-your-life

What are your reasons to choose programming as your profession?

I'm inept at pretty much everything else.

I'd like to order people around, but I find it too difficult to bend them to my will. Whereas computers more or less do what I tell them.

I like abstract thinking, but I'm too stupid for maths and too lazy to write long philosophical essays.

I'm too fat for hard physical work. And don't like to get cold, wet or dirty. Or to hang out in dangerous places.

I'm too claustrophobic to work in coal-mining (which is what generations of my ancestors did)

I hate dressing up and don't have the chutzpah for sales, marketing or similar activities that require an outgoing personality.

That doesn't really leave me with much option. It's programming or filing clerk. And the programmers are putting the filing clerks out of business.

See : http://www.quora.com/Programming-Languages/What-are-your-reasons-to-choose-programming-as-your-profession

Why does the tech culture still churn out so many people who have zealotry about their choices of language, operating system, or other layers of the tech stack? Do others think the negatives outweigh the positives?

No. I think the positives of the zealotry and the "religious wars" outweigh the negatives.

Or rather :

a) people argue because they *care*. And caring is a good thing.

b) what they care about is aesthetic. You can't boil aesthetics down to some easily testable criteria. So you can't easily resolve it. But that doesn't mean it's not important. Or crucial to the human soul.

Ultimately, arguing about programming languages helps programmers think about programming languages and have a deeper and more passionate understanding of their tools. And that's the only way they can become really engaged. And good at what they do.

All good craftsmen care passionately about their tools.

But software is slightly unusual because it's an area where people who need to be craftsmen are very often obliged to work with tools (languages / OSes) that they didn't choose. So of course that creates dissent.

See : http://www.quora.com/Why-does-the-tech-culture-still-churn-out-so-many-people-who-have-zealotry-about-their-choices-of-language-operating-system-or-other-layers-of-the-tech-stack-Do-others-think-the-negatives-outweigh-the-positives

What do you prefer? Curly braces (like in C) or indentation (like in python)? Why?

Python indentation … it's cleaner and less typing.

Indentation is a really useful discipline to have, which gives me a quick visual check of the structure of my program. I'd never not indent, even in a brace language.

As to one space causing a disaster, it never happens to me in practice. I have indenting at my fingertips. Something I got used to it after my first 2 or 3 days with Python and never looked back.

Today I also use CoffeeScript in preference to Javascript. And I'm trying to get my head around Haskell. Indenting and lack of brackets was a big plus in Haskell's favour over Clojure, Scala etc. for me.

See : http://www.quora.com/What-do-you-prefer-Curly-braces-like-in-C-or-indentation-like-in-python-Why

Why and how do you prefer to indent while using curly braces?

Give me 1. any day

It saves me 1 line per function. As someone who attempts a "functional style" even in C (by which I mean I have a lot of very short one or two line functions) I can fit almost 30% more code on a screen compared with 2. That's a significant win, less scrolling and having to keep things in short term memory.

See : http://www.quora.com/Why-and-how-do-you-prefer-to-indent-while-using-curly-braces

What is the hardest thing you do as a software engineer?

Get into "the zone" when debugging.

Writing original code is fun. You can get into it easily.

Debugging something is NOT fun. Once I'm in the zone I can spend hours resolving all kinds of awkward problems. But before the zone, faced with stupid annoyances , when it's tempting to check if there's been any activity on Quora or my favourite news feeds … I can spend far too long faffing without being able to engage my mind with the problem.

See : http://www.quora.com/Software-Engineering/What-is-the-hardest-thing-you-do-as-a-software-engineer

Can everyone become a good programmer or must the person be gifted?

I don't know.

I've taught enough programming to know that some people "get it" and some don't seem to.

I can't say that those who don't initially, might not if they practice harder. I would never say it's genetic, but certainly some people have the temperament and inclination for it. And others don't seem to.

See : http://www.quora.com/Computer-Programmers/Can-everyone-become-a-good-programmer-or-must-the-person-be-gifted

Why is it that when it comes to computer programming, everyone completely warns against doing it for the money?

Computer programming isn't a job. It's a vocation.

See : http://www.quora.com/Why-is-it-that-when-it-comes-to-computer-programming-everyone-completely-warns-against-doing-it-for-the-money

Would you do your job for free if you could?

I already do.

I'm a computer programmer. I've worked in everything from web-startups, to non-profits, to enterprises contracted to the government to teaching programming to university undergraduates. Right now I'm not employed, but I find myself busier than ever, and writing software quite a lot of the time.

Of course, when I write software for myself, it's for projects that I'm interested in, some of which I hope will have some kind of commercial payback in the future. Whereas when I take a salary it's to write software that I wouldn't have necessarily chosen to write. So that autonomy makes all the difference in the world. But the activity is remarkably similar. (Bugs are bugs)

See : http://www.quora.com/Would-you-do-your-job-for-free-if-you-could

Why are people on Quora so upset with terms like "rockstar/ninja programmer"?

I don't personally see anything wrong with the term. You need a word for people who are good developers, and it might as well be "rock star" as anything else. And if it reinforces the idea that good programmers are cool, then fine.

But please recognise that the term is MEANT to be ironic - because rock star programmers are nothing like the stereotype of rock stars. The aren't loud, showy, with an attitude. If the term leads stupid people to think that they can evaluate great programmers based on how much of a braggart they are, then it's doing them a disservice. OTOH stupid people gonna be stupid, right? Whatever words you give them.

See : http://www.quora.com/Computer-Programming/Why-are-people-on-Quora-so-upset-with-terms-like-rockstar-ninja-programmer

Do I still have a future in Programming if I don't do well on the Ap Computer Science Test (tested in Java?).

Here's the honest test :

Now that you "like" programming, are you doing a lot of it? For yourself, your own projects etc? If so, you have a future in it.

If you ONLY do it for classes, when pressurized by project / deadlines etc. you probably don't .

See : http://www.quora.com/Computer-Science/Do-I-still-have-a-future-in-Programming-if-I-dont-do-well-on-the-Ap-Computer-Science-Test-tested-in-Java

How does one stay focused on one programming language?

1) You totally don't want to do that. Programmers should try to learn a new language more or less every year. And a year is enough to learn most things that are important about a language. Of course there are *skills* in programming that take a life-time to master, but they are rarely things which are specific to one language.

2) The best way to get stuck using one language for the rest of your life is to get a corporate job maintaining legacy code.

See : http://www.quora.com/Computer-Programming/How-does-one-stay-focused-on-one-programming-language

I hate computer programming. Is there a role for me in tech or this world at all?

The correct answer is “yes”. But having read all the answers which suggest you go into testing / management / customer support / design etc. I’m starting to think, FUCK NO! There are already enough twats in this world who think that because they can’t program that must mean they have some special insight into telling me what to do.

Go and sail boats or hand out parking tickets or something. Leave us alone. If you’re not in tech for the love of tech it probably means you’re just in it for money.

See : http://www.quora.com/I-hate-computer-programming-Is-there-a-role-for-me-in-tech-or-this-world-at-all

What was it like to be a programmer without the Internet?

We had a lot of magazines which published reviews, source-code of programs (we called them “listings”) and other bits of news and gossip about the computer scene.

We didn’t really know we didn’t have the internet, because we didn’t envisage such a thing as exists today. Most of what we wanted to know we got from the magazines and that seemed an acceptable and viable bandwidth for getting this information (two or three magazine’s worth a month, decided by an editor.)

See : http://www.quora.com/Computer-Programmers/What-was-it-like-to-be-a-programmer-without-the-Internet

Dear programmers: if there were no more computers, what would you do with your life?

aAAAAAAAARGHHH!!!!!!!

See : http://www.quora.com/Dear-programmers-if-there-were-no-more-computers-what-would-you-do-with-your-life

Performance

Are younger programmers better programmers? Why or why not ?

Programming knowledge is cumulative. It doesn't become harder to learn a new language because you've spent 5 years working with an older language.

People who assume this are just wrong.

The more languages you know, the easier it is to learn the next one. When you come across "unfamiliar" concepts or requirements, you often find they *have* popped up before in a slightly different form in some other language or situation.

You pick up things by "*triangulation*" and analogy with existing knowledge.

All other things being equal, older programmers have more experience. And will learn and understand and figure out solutions faster.

However, what you lose as you get older is *stamina.*

Programming is a demanding job. It requires long periods of intense concentration. You are continuously confronted with frustrations : bugs that you can't figure out, management decisions that are wrong-headed, tools (whether hardware or frameworks and APIs) that aren't ideal for what you want to do. You definitely lose tolerance for struggling with these things.

So are young programmers "better"? Better for what?

Are they cleverer or more intuitive? Mostly not. I cringe when I remember how mind-boggling ignorant I was when I was 21 and happily recommending all kinds of idiotic solutions to things. ("Why don't you take your extremely sophisticated Fortran code, written by engineers, which you are successfully *selling* for tens of thousands of pounds, and rewrite it in C++ because C++ is OBJECT ORIENTED and then you wouldn't need to have anything as clunky as a bunch of different Unix programs communicating by writing FILES for each other. You could have all that data represented in a bunch of nested objects in memory! Plus you'd be able to run it on Windows!" Yes I really argued that with my colleagues and yes I really am embarrassed and thankful that they took no fucking notice of me at all.)

Young programmers are certainly not better because they happen to be more in touch and in tune with the latest practices and technologies. There might have been some value to this heuristic 30 years ago when there weren't many channels to propagate knowledge of new tools and technologies, and when CS degrees were the most effective circulators of information.

But today … internet. A 60 year old can follow Lambda the Ultimate and StackOverflow and InfoQ etc. as easily as an 18 year old can. And probably pieces together the big picture better than the 18 year old.

But here’s where the young programmer *does* score. Give her pizza and plenty of coffee and she’ll stay in the office until 2 in the morning for days on end, struggling with bugs, digging into the ephemeral quirks of the API, fascinated and delighted that someone cares about her work.

In contrast, the 40 year old wants to get back to her husband and kids by 9pm. She’s seen a dozen APIs come and go. She’s not the slightest bit interested or curious about how THIS particular one works. She’s just pissed that it doesn’t work the way she imagined it did yesterday, because now it’s wasting her time.

She’s not going to donate as much of her spare energy to you, or to solving yet more problems of accidental complexity that this particular project has thrown up. She’ll be more likely to complain when management demand she goes on yet another death-march.

Programming needs energy and stamina. And young people have a lot of both. Programming is also intensely intellectual so experience and wisdom do count for a lot. And it’s largely contingent on the domain and the kind of programming being done whether energy trumps wisdom or vice versa.

See : http://www.quora.com/Are-younger-programmers-better-programmers-Why-or-why-not

What skills do self-taught programmers have that others don’t?

I’m not convinced that there are any programmers who AREN’T self-taught in some sense.

No course or text-book is going to cover even a tenth of the material you use the moment you get out there and start trying to engage real systems.

Not all programmers are taught, but no working programmer isn’t also self-taught

See : http://www.quora.com/What-skills-do-self-taught-programmers-have-that-others-dont

What are the ways to excel in programming, given having only a little knowledge in coding so far?

```
        Write lots of programs.&#160; It's as simple as that.
```

It’s not too late to become a great programmer. But there is no short-cut that avoids doing a lot of programming. (Eg. you can’t read books or websites as a substitute). Just get off Quora and start writing programs. ## How can you know how many hours of programming my idea is going to need?

You can’t.

End of story.

See : http://www.quora.com/How-can-you-know-how-many-hours-of-programming-my-idea-is-going-to-need

How long does it take programmers to reach a state of ‘flow’?

It can take the whole morning to get started.

Once I’m in the flow there’s some momentum. Minor interuptions don’t necessarily knock me out of it. I can deal with them and re-engage. But a big interuption … that’s probably the rest of the day gone.

See : http://www.quora.com/How-long-does-it-take-programmers-to-reach-a-state-of-flow

How does one go beyond introductory computer science/data structures and start building programs and apps on their own?

If you want to get good at writing programs, write programs. There’s no “theory” which substitutes writing code. There are theories about writing large-scale code. But you won’t *really* understand them until you’re actually trying to do it.

“Designing” without coding is an empty exercise because you won’t be getting feedback from your computer about whether the design is working or not.

I’m not saying design is useless. But it does need to be tightly integrated with the programming process.

Secondly, write the software you WANT to write and to have. Exercises set by someone else are OK. But they’re also pretty boring and writing code requires a lot of self motivation. So it’s a lot easier to push yourself through it if you are building something where you *want* the end result.

So. Pick an application that you wish existed: it could be a website that lets you share some information with your friends. It could be a phone app. that would be useful to you. It might be a game you want to play.

Now try to figure out how to make it reality.

Just START. Even though you don’t have any resources or knowledge other than from the small scale exercises you’ve done. It doesn’t matter. Even if it doesn’t turn out well to begin with (which it won’t), you WILL learn. And that’s the real goal here.

So, what information will your program need to store? What kind of data-structure will you need in order to store that information? Will you need files on disk? A database? Look up on the internet how you'll open and write a file. Look up how you'll install, configure and open up a connection with a database.

Will it need a graphical user interface? In the browser? On the desktop? In the phone? Fine, find out how to do that. In HTML / Javascript in the browser. Or with the standard APIs for the platform.

Does it need the data to be presented in alphabetical order? Look up sorting algorithms.

Etc. Etc. Etc.

Tools will help answer these questions. And an IDE may have a plugin to design the GUI. Or to configure the database. But you won't know *why* the tool is important or understand how to use it UNTIL you come across the problem that needs it.

So don't start by looking for tools or techniques. There are no "secrets". Start by looking for the problem you want to solve and then trying to build the program that solves it. That is the only path to understanding everything else.

See : http://www.quora.com/How-does-one-go-beyond-introductory-computer-science-data-structures-and-start-building-programs-and-apps-on-their-own

Automation

Is programming going to be automated in the near future? If so, is it worth going into computer science?

Programming, basically, *is* the art of automation.

So programmers are always in the business (http://www.quora.com/Business) of automating away some of their (boring) tasks. That's why we have compilers to write machine code for us and smart editors to correct our syntax errors.

But we'll never automate the activity of automation itself. Because humans will still be the ultimate deciders of what we want computers to do for us. And somebody still needs to make the hard choices that specify *exactly* what any particular mechanised system will do. And those people will be programmers.

Programming won't go away because that need - translating between informal human desires and mechanically executable formal requirements - won't disappear.

See : http://www.quora.com/Is-programming-going-to-be-automated-in-the-near-future-If-so-is-it-worth-going-into-computer-science

What is the simplest analogy to explain why computers cannot automatically write code and program by itself for the programmer?

If I run a shop, why I can't I be my own customer? Then I could guarantee I'd get a lot of sales.

See : http://www.quora.com/Computer-Programming/What-is-the-simplest-analogy-to-explain-why-computers-cannot-automatically-write-code-and-program-by-itself-for-the-programmer

When a program doesn't compile, error messages are sometimes esoteric. Often a simple Google search leading to a site like stack exchange solves the problem. Now why can't we automate this? How about a Siri for coding?

I definitely believe that compiler and other kinds of debugger / IDE error messages could be improved. But Siri-like assistants certainly haven't been proved as more than gimmicks.

They contain relatively little, and relatively stereotypical information that's based on a fairly simplistic model of what the user wants. Programmers' interactions with the computer are a much richer and more complex domain.

Ideally, an intelligent programmer's assistent would have to model a LOT more of what the programmer is doing and thinking. I think we'll move in that direction, but the fruits of it will appear in the form of better IDE tools rather than a big jumpt to "intelligent assistant" type tools.

See : http://www.quora.com/Computer-Programming/When-a-program-doesnt-compile-error-messages-are-sometimes-esoteric-Often-a-simple-Google-search-leading-to-a-site-like-stack-exchange-solves-the-problem-Now-why-cant-we-automate-this-How-about-a-Siri-for-coding

Will programming become obselete within the next 50-60 years?

First see Phil Jones' answer to What are some of the most common misconceptions/myths about programming? (http://www.quora.com/Computer-Programming/What-are-some-of-the-most-common-misconceptions-myths-about-programming/answer/Phil-Jones)

What's hard in programming is not mastering the syntactic vocabulary of programming, but designing algorithms and expressing problems in a "mechanizable" way.

I don't believe that these two problems will disappear, even if we supplement them with two further modes of instruction : directly teaching computers and robots things by example, and giving computers fuzzier natural language descriptions of what we want. Both of these will work within implicit "frames" of context but we'll still need programming to cross the boundaries of such contexts. Or to define completely new contexts. Or to specify in exact detail what we want things to mean.

A lot of automation "makes things easier" by offering a preset menu of options. This commoditisation is a very useful part of making the power of computing more accessible. But it will never provide people or companies "the edge", because, by definition, the edge over your competitors is the bit that hasn't been commoditized yet. Where someone has to have the idea of doing something, or of how to do something, completely new. That activity will still need developer intervention to bring about. And mostly it will involved plugging things together that have never been plugged previously. Or filling and processing data-structures that have never been previously filled / processed. Both activities will ultmately involve programming.

See : http://www.quora.com/Programming-Languages/Will-programming-become-obselete-within-the-next-50-60-years

What will become of software engineers once computers learn to code themselves?

Computers won't learn to "code themselves" because the whole point of programming is to match what computers do with what humans WANT them to do.

Programmers will always be necessary to make explicit for the computer what the humans want.

Sure, the tools will get better. Languages will get higher level. Programming will get "easier". But it never goes away.

Even today you still can't read with 100% accuracy what the person you live with really wants. Similarly, no computer will ever be able to do that. It's not just about "AI completeness". It's about the fact that there actually isn't such a thing until we decide to explicitly say there's such a thing.

And that's why society will always have a role for someone to make its desires explicit. And those people will be programmers.

See : http://www.quora.com/What-will-become-of-software-engineers-once-computers-learn-to-code-themselves

Predictions

What is the future of software engineering in 2020: harder or easier development?

I'd predict three trends for software engineering in the next 6 or 7 years.

1) "Reactive" programming. A new generation of languages will do what Angular.js etc. do : let you declaratively define what data you want and under circumstances it updates without having to think about the flow of control or the call-backs explicitly.

2) That will blend into Business Process Modelling tools and tools for Sys-Admin of virtualization / clouds. People we'll get languages that declaratively describe how a bunch of systems should talk to each other at the high-level and many programmers will be able to forget about fiddling around at the detail of individual messages between the browser or app. and the server. There'll just be schema to describe the kinds of data the overall system stores, where it has to be, and when it has to be updated.

3) Increasing interest in the internet of things. There are going to be a lot more embedded devices, sensors, small wireless meshes and robots. People used to the convenience of high level languages on the web aren't all going to want to learn C and think at such a low level. So expect that we'll get data-flow / reactive / event handling languages on the Arduino etc.

So fiddly low-level communication is going to get easier. But we'll be trying to orchestrate larger and more complex swarms of machines which will take a certain kind of thinking.

See : http://www.quora.com/What-is-the-future-of-software-engineering-in-2020-harder-or-easier-development

What concepts from programming language research are likely to become widespread in the next 5 to 10 years?

I think "Reactive Programming" (or FRP in its functional version) will become this decade's "garbage collection". (Ie. standard in all languages that don't have a particularly good reason not to have it.)

Basically a way to wire together values and have them updated automatically when some events occur. Without having to explicitly write the event-handling as nested event / callback handlers. We've had this idea for ages in various forms, as "signals" in databases and web frameworks, in browser-side frameworks like angular.js. But I think it will become a first-class citizen of commonly used programming languages.

The frameworks show that many languages can implement RP in a library / framework. To become widespread, we'll need some kind of syntactic support (eg. to distinguish between one-shot function calls with particular parameters and permanent data-flows) And good compiler / debugger support because debugging spooky action at a distance in languages which aren't prepared for it, sucks.

Many FP languages already have something similar. I'm guessing we'll see the first attempt to graft it into the javascript or java standards this decade.

See : http://www.quora.com/Programming-Languages/What-concepts-from-programming-language-research-are-likely-to-become-widespread-in-the-next-5-to-10-years

What will the next generation OS look like?

Distributed.

It will be an OS for a swarm of tightly connected devices. Watches, tablets and any local large TV screens in your area will act as output devices. Sensors and cameras will have equal status to keyboards, touch and mice as input devices.

It will have to seamlessly integrate all the devices you carry on your person, home-automation sensors and actuators, cloud-based storage and an app-store / package-manager model for installing software and granting it permissions over all these things.

Permissions and parallel processes will be crucial. Time and synchronization too. Individual "programs" will have to be running at multiple sites : on the server, on your laptop and on your watch.

Programming languages / frameworks will have to manage this transparently (things like meteor.js with its transparent syncing between browser and server are a start). "Functional Reactive Programming" ideas of setting up automatically updated flows of data will become a standard part of the wiring. But the programs will have to be able to orchestrate these flows across devices. And cope with outages. And errors coming down the pipeline.

See : http://www.quora.com/What-will-the-next-generation-OS-look-like

What will Computer Science and Programming be like in the far future?

There's no programming language that's good for everything. Not even "natural language". We invented maths notation so that people who are "natural intelligences" could communicate more precisely and effectively about certain abstract things.

No profession, from medicine to law to theoretical physics to art criticism is without its jargon which can only be understood by those who've taken the time and effort to achieve a certain level of familiarity with it.

Programming *won't* therefore become something that untrained people do by conversing in everyday language, for the simple reason that most programming will still be about involving the computers in specialist activities, and will need specialist understanding of those activities. Such specialist understanding actually prefers more formal and unambiguous notation rather than trying to use conversational natural language. (Even ideas and services that everyone uses and thinks they understand (think banking, or Facebook) still have complex conceptual ideas specifying how they actually work and deal with edge-cases behind the scenes. You would be hard pressed to specify how your bank or Facebook work to another human being, despite that human being being "AI complete".)

So, most likely, programming will fragment into more and more "domain-specific languages" for the increasing number of niches that are programmable. Each will be full of domain-specific assumptions and knowledge, though they may sometimes be different dialects of some basic grammatical syntactic patterns. (An example of this is XML which acts as a syntactic substrate to many different data formats.)

Apart from XML, C and Lisp are syntaxes which will probably still be hanging around. There'll be many of these specialist languages that still use C-like or Lisp-like notation because that's what people are used to.

The other thing about these domain-specific languages is that many of them will be hooked into huge online data-bases and cloud hosted services. (A bit like the Wolfram Language). You'll rarely write code to run by itself as much as you'll use it to present your problem to the cloud service and reformat the returning data.

There'll still be "serious" programming languages for defining and orchestrating what goes on behind the scenes. Such languages will have the following characteristics :

- they'll be good for defining and hosting these other DSLs.
- they'll have facilities for defining data-flow between systems. (Most likely some kind of reactive programming). Because the important thing about programming in the future is that you'll very rarely be programming one computer. You'll be programming an application that's distributed amongst many : servers and phones, the sensors and actuators of the "internet of things", multiple robots etc.
- they'll aim to be as powerful, expressive and error free as possible.

(If all of this leads you to think of Haskell, that's deliberate .. .though it may not be Haskell itself.)

See : http://www.quora.com/What-will-Computer-Science-and-Programming-be-like-in-the-far-future

The Craft

What are some real life bad habits that programming gives people?

Terrible sitting posture. I'm sure my back is going to collapse on me one of these days.

See : http://www.quora.com/What-are-some-real-life-bad-habits-that-programming-gives-people

Why are all programming languages difficult to configure with its appropriate environment?

Because that's actually one of the hard problems.

There are two conflicting requirements :

1) in order for the program to do real work, it needs to be able to access the machine's resources : memory, disk, mouse and keyboard, networking, sound-card, USB, GPS etc. etc.

2) in order to be relatively platform independent, (which allows programmers to be reuse their knowledge across platforms and operating systems etc.), the language has to be fairly self-contained and loosely coupled to the platform.

To satisfy both requirements you need some kind of *mapping* between the resources represented virtually within the programing language, and the underlying machine and operating system.

And that mapping is done via a configuration which is *external* to the main program. (It has to be external, if the program could see the machine resources directly it wouldn't be portable between different platforms.)

Because the configuration is external it's not very visible to the programmer. It's necessarily outside the world of the programming language, and usually outside the programmer's comfort zone. And, naturally, you can't really write portable tooling to help manage it. So tooling for managing and debugging this configuration is usually somewhat inferior to other libraries and resources (like editors).

BTW : this is a universal principle, equally applicable for "virtual machine" languages like Java, and for compiled languages like C that just have a portable standard library.

The only languages that *are* "plug'n'play" are languages which don't have the two requirements.

Either they're tied to a single platform : Visual Basic on Windows or in Office. Perhaps Objective-C for iOS.

Or they're languages that don't offer much access to the underlying platform : Javascript in the browser. Perhaps Smalltalk most of the time. And other languages that stick to their "toy" environments.

The moment you want portability as well as access to the full range of machine resources, you are stuck with having a mapping layer that needs to be configured independently of your actual program, using obscure and non-standard tools and knowledge.

See : http://www.quora.com/Why-are-all-programming-languages-difficult-to-configure-with-its-appropriate-environment

Unit Testing

What are good ways to avoid bugs while programming?

Funny to see Kent Beck (http://www.quora.com/Kent-Beck)'s answer here. Because he's the guy that actually SOLVED the problem of bugs.

Or rather, every programmer knows you can't avoid making mistakes in programming. What you CAN do, as Beck taught us, is to minimize the size and difficulty that most bugs present, by writing your code in fine-grained, tight feedback loops where you write a very small amount of new code against a very small piece of code designed to test that this new code does the thing you expect. (The technical name for this is "unit-test")

By running automated tests for every 3-5 lines of new code you write, you usually know that the bug you just discovered is localized within the few lines you just wrote. And it's pretty straightforward to track it down there. If, OTOH, you write 300 lines of code either without testing or with desultory manual testing, you'll find afterwards that those 300 lines are full of new bugs that are going to be HARD and time-consuming to track down.

Fans of strong typing like Tikhon Jelvis (http://www.quora.com/Tikhon-Jelvis) will point out that compile-time type-checking is another way of getting early / fast feedback on errors (which also helps to localize them.) Here, certain kinds of bugs never get into your running code at all because the compiler catches them.

Both of these approaches : TDD and strong typing are ways to help reduce the amount and harm of bugs in your code. Neither is a silver-bullet which works perfectly and without cost : Phil Jones' answer to Software Testing: Why do so many developers not deliver unit tests with their code?</a>
,

See : http://www.quora.com/What-are-good-ways-to-avoid-bugs-while-programming

What's the best way to begin unit testing a years-old system that is currently only integration tested?

It depends.

I confess, that I sometimes just start writing unit-tests and rebuilding components in a test-driven way from scratch. You can use the existing component to generate the test-data. Then at some point you just switch over.

Caveats. Yes, I know there are lots of reasons NOT to do this. And you wouldn't do it on too large a component. I just want to say that I've gotten away with it sometimes.

See : http://www.quora.com/Whats-the-best-way-to-begin-unit-testing-a-years-old-system-that-is-currently-only-integration-tested

Why do so many developers not deliver unit tests with their code?

I'm a big fan of unit-testing. But the hardest thing about unit-testing is that hand-generating or independently generating the intended results of code can be non-trivial.

It's easy enough to write a unit-test for a bit of code that does basic integer arithmetic. You do the arithmetic in your head as you're writing the test.

But what about a unit-test for a function that finds the length of a vector?

You'll almost certainly think … (and I confess, I've done things like this) … hmmm .. vector [10, 0, 0] should have a length 10 shouldn't it? I'll test that case.

You may or may not try a test of something like the length of [5, 4, 9] because, unless you pull out your calculator you don't know what that's meant to produce.

And what about a function to rotate a cube in 3D space? Whoah! Floating point matrix multiplication on 6 points. Doing that by hand (even with a calculator) will take … what … 15 minutes? Half an hour? Especially if I have to look up how to do matrix multiplication by hand because I always forget the order.

But now, my function which ray-traces the cube into a bitmap. WTF? How the hell am I going to generate the 1000 X 1000 bitmap data to compare my function's output with?

(OTOH, I can just look at my bitmap output on the screen and tell in a few seconds if the function worked or not.)

And this isn't just for graphics. It's for any serious heavy-lifting data transformation. I've written templating systems that generate HTML and other file formats. And I've spent so much time tracking down "failures" in my tests that are simply due to too many (or too few) carriage-returns in my example correct output. It can take longer to debug your test data than write the function.

Of course there are tricks. You use previously unit-tested functions to help generate the results for the next test. Which is good. But you can sometimes find yourself pretty much copy and pasting the same code between the fixings that generate the required test-results and the main code that actually passes the test. (Which is not really so good.)

Or you'll grab a snapshot of the output of your function (which "seems to be working") and store that. (At least the test will discover if you break things in future.)

Or you'll test derivative attributes of your data. (I can't really test if the values of this million item list are correct but I'll at least test that there are a million of them.)

Sooner or later, as you start working on code that works with more complex data-structures, you'll notice that your tests are becoming a) more expensive to write, and b) less comprehensive in their coverage.

At some point, you'll have to declare bankruptcy : "to generate comprehensive test data is going to take next week, but the results look OK today".

OK. Having said all that, don't be discouraged. Unit testing and test-driven development is an absolutely excellent thing. If you can do it, you will produce better code, faster, and you'll spend far less of your time in future breaking and fixing your breakages. KIDS! DO UNIT TESTING!

See : http://www.quora.com/Software-Testing/Why-do-so-many-developers-not-deliver-unit-tests-with-their-code

Listings

Why is literate programming so unpopular?

tl;dr It doesn't really solve the problems people have as well as other tools and strategies that are more popular.

The basic idea of literate programming is that documentation is such a good thing that you shouldn't keep it separate from the program but embed the program into the documentation.

The contrary truth is that documentation isn't as wonderful as Knuth assumed. And there are other ways to combine "documentation" with code that have proved more congenial to people, and more effective.

Firstly, well written code ought to, to a certain extent, be clear enough for another programmer to read and understand what's going on. That is the most important thing. Adding natural language descriptions of what your code is doing, alongside that, is a very poor substitute for readable code.

If your code is well written, the next programmer won't need to read a long commentary on it because he or she will be able to follow your code directly.

And if your code isn't well written then

a) it's unlikely that your attempt to write a natural language description of what it does is much better.

b) even if your description is wonderfully clear, then the next programmer is still going to have to struggle with your badly written code, just to be able to make sense of the mapping between the clear documentation and the unclear code.

Now he / she has two problems.

However, this should all be understood in a context where a lot more human-readable labelling has moved into programming languages themselves. When Knuth was promoting Literate Programming people were still using languages where you wrote :

LET X = 10 GOTO 5000

And if you didn't comment or document what was going on at line 5000 then the next programmer would have to read another 100 lines of low-level code to know why you even did that.

Today, you're more likely to call a function

sortedUsers = quicksort(userTable)

which despite being genuine executable computer code is also 95% human-readable labelling.

So in that sense, documentation DID get integrated with code. But in a different way from the way LP imagines. Languages got a lot higher level (can hide a lot of the computer-oriented *mechanism* from the programmer), and code now has a far higher proportion of human-readable tokens than it did previously.

Beyond that, the web happened. Suddenly we had access to all the documentation we needed at the click of a hyper-link. When I read someone's code that calls an unfamiliar API I don't need the original programmer to have documented what that API call does for me. Because I can just type two words into Google and see it all. I can see the official documentation. I can see examples of people using it. I can see discussions where people complain about problems with it and suggest workarounds. There's no way that the original programmer, however diligent in his or her literate documentation, could provide a fraction of that value to me.

Finally, when Knuth was promoting Literate Programming, people were still printing out listings (http://www.quora.com/Is-it-unconventional-for-a-programmer-to-sift-through-lines-of-lengthy-code-printed-on-paper-rather-than-on-his-her-screen-in-order-to-find-bugs-and-logical-errors-Why/answer/Phil-Jones), to read them *offline,* and then typing in their modifications (sometimes on punched cards). In that world, Literate Programming made sense. All that you had to work with was the listing that came off the line-printer. Any information that wasn't in that listing wasn't really available to you. So the more information that was in it, the more the programmer explained his or her thoughts and rationales, the better.

That's not a world any of us live in today. Today if we want to know what something does we can type it on the REPL or our IDE lets us drill down to the definition in a library. Our IDEs auto-complete the correct names of things for us. We develop against automated tests which continuously alert us to problems. This is a much faster, more fluid, interactive way of developing software. We develop in a live "conversation" with the computer. We rarely have chunks of time when we step back from the machine and read and contemplate and simulate the running of an algorithm in our heads.

So Literate Programming is a solution for an age which has passed. When the tools were different. When the roles of programmers and computers were different. When documentation was scarce and very hard to access. And the languages were so low level that most of a programming listing was in machine-speak not human-speak.

Today's programming challenges need solutions specific to them.

See : http://www.quora.com/Literate-Programming/Why-is-literate-programming-so-unpopular

Is it unconventional for a programmer to sift through lines of lengthy code printed on paper rather than on his/her screen, in order to find bugs and logical errors? Why?

It used to be the norm. But today you should probably take it as a “smell” (ie. an intangible warning that something is going wrong.)

What’s wrong is printing and reading code on paper implies working on too large chunks of your code at the same time.

Ideally you should work at a fine granularity : for example, write short functions (for me that’s 5-10 lines max, depending on language) which are “self-evidently” right or passing their unit-tests.

That rhythm … write unit test, write or modify function to pass it, refactor to eliminate redundancy, should ideally start to get quite quick, and start to flow.

But printing a couple of pages on paper doesn’t fit into that rhythm. It just takes too long. And if you have to print out a couple of pages of code to sift through them to find your bug, it’s probably because that’s the granularity you’re working at : chunks of two pages where there’s a bug hidden somewhere but you don’t know where. That’s looking for a needle in a haystack!

Now it’s not that you can avoid the situation entirely, but ideally, if you’re working at a finer granularity, using unit-tests, you should very rarely hit the problem of “I have two pages of code but I don’t know where my bug is”. It should be more like “This latest function I wrote doesn’t pass the latest unit-test so either one of these two *lines* is wrong or one of the three functions they depend on is wrong.”

In which case you double check the two lines you just wrote and if you can’t see a problem there, you write a couple of extra unit tests for the other functions, to make sure that they are delivering what’s expected in this situation. And you follow the bug back up the calling tree of functions, closing it down with those extra unit-tests.

When you’re writing like this - and I recommend it as the best way to write code - even in bad cases of very obscure bugs, your debugging propagates back up the tree of calling functions. It doesn’t involve whatever two pages of lines just happen to be arbitrarily contiguous in the file you’re working in. But that’s what printed listings are going to give you.

If you have a lot of reuse in your code (which is generally a good thing) then you will almost certainly have dependencies at a distance. And so what do you do then? Print 20 pages of ALL the files in your project in order to track the bug through them? Or do you let fear of having to trace a bug across several files PREVENT you from having too much code reuse (thus letting unnecessary redundancy into your program)? It’s an awkward mismatch.

So printing code isn’t a bad in itself but it should set off a lot of warning bells.

See : http://www.quora.com/Is-it-unconventional-for-a-programmer-to-sift-through-lines-of-lengthy-code-printed-on-paper-rather-than-on-his-her-screen-in-order-to-find-bugs-and-logical-errors-Why

Is is unconventional for a programmer to print out pages of lengthy code on paper rather than sifting through line after line on his/her screen to find bugs and logical errors?

It used to be the norm. But today you should probably take it as a “smell” (ie. an intangible warning that something is going wrong.)

What’s wrong is printing and reading code on paper implies working on too large chunks of your code at the same time.

Ideally you should work at a fine granularity : for example, write short functions (for me that’s 5-10 lines max, depending on language) which are “self-evidently” right or passing their unit-tests.

That rhythm … write unit test, write or modify function to pass it, refactor to eliminate redundancy, should ideally start to get quite quick, and start to flow.

But printing a couple of pages on paper doesn’t fit into that rhythm. It just takes too long. And if you have to print out a couple of pages of code to sift through them to find your bug, it’s probably because that’s the granularity you’re working at : chunks of two pages where there’s a bug hidden somewhere but you don’t know where. That’s looking for a needle in a haystack!

Now it’s not that you can avoid the situation entirely, but ideally, if you’re working at a finer granularity, using unit-tests, you should very rarely hit the problem of “I have two pages of code but I don’t know where my bug is”. It should be more like “This latest function I wrote doesn’t pass the latest unit-test so either one of these two *lines* is wrong or one of the three functions they depend on is wrong.”

In which case you double check the two lines you just wrote and if you can’t see a problem there, you write a couple of extra unit tests for the other functions, to make sure that they are delivering what’s expected in this situation. And you follow the bug back up the calling tree of functions, closing it down with those extra unit-tests.

When you’re writing like this - and I recommend it as the best way to write code - even in bad cases of very obscure bugs, your debugging propagates back up the tree of calling functions. It doesn’t involve whatever two pages of lines just happen to be arbitrarily contiguous in the file you’re working in. But that’s what printed listings are going to give you.

If you have a lot of reuse in your code (which is generally a good thing) then you will almost certainly have dependencies at a distance. And so what do you do then? Print 20 pages of ALL the files in your project in order to track the bug through them? Or do you let fear of having to trace a bug across several files PREVENT you from having too much code reuse (thus letting unnecessary redundancy into your program)? It's an awkward mismatch.

So printing code isn't a bad in itself but it should set off a lot of warning bells.

See : http://www.quora.com/Is-is-unconventional-for-a-programmer-to-print-out-pages-of-lengthy-code-on-paper-rather-than-sifting-through-line-after-line-on-his-her-screen-to-find-bugs-and-logical-errors

Issues

What famous computer science quote essentially says “bad ideas win”?

Toby Thain (http://www.quora.com/Toby-Thain) is probably right that “Worse is Better” is what the questioner was thinking of.

But “bad ideas win” is not really what Worse is Better actually says / means.

What WiB is really trying to point out (IMHO) is that certain traditionally exalted computer science ideals needed to be balanced against rather prosaic implementation considerations.

So in his canonical example, Gabriel talks about how a reusable component should have a clean, consistent interface that doesn’t leak problems. That’s the kind of thing we are all taught is a virtue in computer science. AND IT IS TRUE.

But he then points out that that this may incur a greater implementation and maintenance cost, as the external simplicity requires the component to be more complex internally to handle the error cases. (Think about the way Java’s checked exceptions protect the caller but add a great deal of verbosity to any method that might trigger them.)

Sometimes, success or failure of something depends as much on that level (the slog-work of the implementation) as on the elegance or formal correctness of the abstractions.

All real-world software development involves a negotiation between honouring the abstract and the concrete. And Worse is Better is really meant to be a corrective to the assumption that the virtues of the abstract / ideal world always dominate in its relationship with the concrete world of the materiality of software. (That concrete materiality includes programmer time, intelligence, comfort etc., not to mention performance, hardware costs etc. etc.)

Of course, this is an essay aimed at computer scientists and users of sophisticated and abstract languages like Lisp who may be at risk of prioritizing the abstract over the concrete. Most software development is in corporate environments where the concrete is already highly prioritized and probably abstract ideals are not honoured nearly enough.

Nevertheless, Worse is Better is a very valuable tool in our tool-kit for thinking about programming.

See : http://www.quora.com/What-famous-computer-science-quote-essentially-says-bad-ideas-win

Misc Move Me

What characteristics of a programming language makes it capable of building very large-scale software?

The de facto thinking on this is that the language should make it easy to compartmentalize programming into well segregated components (modules / frameworks) and offers some kind of “contract” idea which can be checked at compile-time.

That’s the thinking behind, not only Java, but Modula 2, Ada, Eiffel etc.

Personally, I suspect that, in the long run, we may move away from this thinking. The largest-scale software almost certainly runs on multiple computers. Won’t be written in a single language, or written or compiled at one time. Won’t even be owned or executed by a single organization.

Instead, the largest software will be like, say, Facebook. Written, deployed on clouds and clusters, upgraded while running, with supplementary services being continually added.

The web is the largest software environment of all. And at the heart of the web is HTML. HTML is a great language for large-scale computing. It scales to billions of pages running in hundreds of millions of browsers. Its secret is NOT rigour. Or contracts. It’s *fault-tolerance*. You can write really bad HTML and browsers will still make a valiant effort to render it. Increasingly, web-pages collaborate (one page will embed services from multiple servers via AJAX etc.) And even these can fail without bringing down the page as a whole.

Much of the architecture of the modern web is built of queues and caches. Almost certainly we’ll see very high-level cloud-automation / configuration / scripting / data-flow languages to orchestrate these queues and caches. And HADOOP-like map-reduce. I believe we’ll see the same kind of fault-tolerance that we expect in HTML appearing in those languages.

Erlang is a language designed for orchestrating many independent processes in a critical environment. It has a standard pattern for handling many kinds of faults. The process that encounters a problem just kills itself. And sooner or later a supervisor process restarts it and it picks up from there. (Other processes start to pass messages to it.)

I’m pretty sure we’ll see more of this pattern. Nodes or entire virtual machines that are quick to kill themselves at the first sign of trouble, and supervisors that bring them back. Or dynamically re-orchestrate the dataflow around trouble-spots.

Many languages are experimenting with Functional Reactive Programming : a higher-level abstraction that makes it easy to set up implicit data-flows and event-driven processing. We’ll see more languages that approach complex processing by allowing the declaration of data-flow networks, and which simplify exception / error handling in those flows with things like Haskell's "Maybe Monad".

Update : Another thing I'm reminded of. Jaron Lanier used to have this idea of "Phenotropic Programming" () Which is a bit far out, but I think it's plausible that fault-tolerant web APIs and the rest of the things I'm describing here, may move us closer.

See : http://www.quora.com/Programming-Languages/What-characteristics-of-a-programming-language-makes-it-capable-of-building-very-large-scale-software

In light of Chomsky's formal language theory, how are the formal elements of programming and language alike?

A2A : I'm not an expert, but I believe Chomsky gives us

a) an understanding that a *grammar* is not just a set of rules that you should adhere to when you write, but a set of rules from which you can generate or test the correctness of any sentence in a language.

b) a classification of different types of languages in terms of the properties of their grammars.

This is important because computer programming is nothing but describing how things can / should be done using language (ie sentences composed of words and symbols.)

A formalization of what a language is, via the grammar which underlies it, makes it possible for machines to work with and on language. This formalization is necessary so that we can write a "parser" (ie. a program which takes instructions written in a language and decodes them into the fine-grained instructions that a CPU can follow.) Parsers are also used in other tools for working with language, such as the automatic syntax checking and colouring in your editors etc.

See : http://www.quora.com/In-light-of-Chomskys-formal-language-theory-how-are-the-formal-elements-of-programming-and-language-alike

How much of what you learned in school do you use in industry?

I was a lousy student. It's taken me over 20 years since college to rediscover a bunch of ideas that were on offer but I was too stupid / lazy / distracted to get.

Things that stuck and I did use : relational database modelling and normalization (SQL was by far the most immediately practical thing I picked up in college was using within a year of leaving). What OO was, and the fact that Smalltalk was cool. The fact that Hypercard-like GUI design environments were cool. A small amount of how computer graphics worked. A small amount of logic. A small amount of graph-theory. A small bit about networking. (Only actually used when I try to explain to someone how the internet works.)

Things that kind of stuck but I didn't use. A rough idea of how a microprocessor works. How to design digital logic circuits. Small amount of VLSI design. That Occam was cool. A bunch of formalizations that describe user-interface analysis and design, but which seem utterly unrelated to real Usability / UX work.

Things I learned that were counterproductive. : that C is horribly complicated and too difficult to use. That "software engineering" was a fascist plot to turn free-spirited programmers into mindless cogs in a giant machine. (Admittedly this was my interpretation of the SE class; and it *is* kind of true.) That formal proofs and specifications were part of the aforementioned plot.

Things that were on offer but I totally failed to pick up on. How to write Lisp. (I never understood what the point was.) How to write Prolog. (I thought Prolog was cool, but just couldn't do it.) What the hell the point of ML was and why anyone would be even the slightest bit interested. What parsing was and how to write a parser (this is probably my largest regret). How compilers work and how to write one.

See : http://www.quora.com/Computer-Programming/How-much-of-what-you-learned-in-school-do-you-use-in-industry

What's easier, being a programmer at a very small company or being a programmer at a large company?

Simon Kinahan (http://www.quora.com/Simon-Kinahan) nails it.

Big companies can be more demanding than small companies. But the stress in small companies is more painful. Normally in big (or even just companies that think they're big / have a corporate mind-set) things are so disfunctional that you tend to feel you're doing a good job relative to the norms. In small companies I've been in the position of *knowing* I was doing a bad job. And that was far far more stressful.

See : http://www.quora.com/Software-Engineering/Whats-easier-being-a-programmer-at-a-very-small-company-or-being-a-programmer-at-a-large-company

What are the advantages of dynamic scoping?

```
        Oh. My. Fucking. God. No. ... No! No! No! No! None! None whatsoever! Zip\
! Zilch! Negatory!
```

Dynamic Scoping is the nearest thing to hell in a programming language. Please don't make a programming language with dynamic scope. And try to avoid using one. For your own sanity.

Particularly don't try working on a code-base someone else has already written in a language that has dynamic scope.

The biggest problem of dynamic scope is that it makes it impossible to refactor and clean up old existing code. You can't look at some code, identify something which is being done badly and replace it with a cleaner / better version. Because you NEVER know if something else was dependent on a side-effect of that old / bad code. So you more or less have to leave ALL the cruft in your program. Forever.

A code-base in a dynamically scoped language is essentially unmaintainable in a way which makes other notions of "unmaintainable" look trivial.

So what are the potential advantages?

Well, it's easier to implement a language with dynamic scope than with lexical scope.

And I suppose dynamic scope gives you a tool to address the hardest problem in programming: dependency injection. Dependency injection sort of becomes trivial when you have dynamic scope. Just have a function somewhere that sets up the variables that define your policies and refer to them everywhere else in your code. Except it's still horrible. ## What are some useful computer related technical skills I can learn within a day?

Yet another snippet extension (http://capitaomorte.github.io/yasnippet/) for emacs.

See : http://www.quora.com/What-are-some-useful-computer-related-technical-skills-I-can-learn-within-a-day

What becomes of our stigmata martyrs when the gods we create and destroy are no longer made in our own image, but in the image of our own creations?

"The Singularity" has been likened to one kind of religion. People have a sort of irrational belief in this moment when computers become "smarter" than us and "everything becomes unpredictable".

William Gibson's "Neuromancer" trilogy ends when the powerful "Artificial Intelligences" that have been guiding the humans in the story fragment and become indistinguishable from Voodoo Loas. (Or something like .)

Even now we're increasingly using mystical and religious terminology for technology.

Here's a quote from the Structure and Interpretation of Computer Programs (https://mitpress.mit.edu/sicp/full-text/book/book.html) :

Mike Kuniavsky explains :

I track bits and pieces of this stuff :

The bottom line is that ever since humans invented language and storytelling they've been fantasizing that language and storytelling might have real *power* in the world. What if stories become sentient? (Gods). What if words alone could create actions and things? (Magic spells). What if ordinary objects could be vested with such powers? (Cauldrons, broomsticks, magic rings etc.)

And now, suddenly, in the age of advanced automation, we're configuring our world around these fantasies. Computer programming IS just using words to create actions. And, increasingly, words to create things.

We call long running processes on Unix "daemons". We learn the right incantations of the APIs to Facebook and Google and Twitter so that we might ask for their favour. We worship them while, increasingly, fearing their wroth.

We have magic books and magic slates in our pockets on which we can read and write anything and which allow us to talk across the world. We're building an internet of things in which all our everyday objects will be enchanted.

We are making the world of advanced automation in the image of that ancient fantasy of a potent language.

I'm not sure, yet, I'm seeing stigmata gods or "god-as-sacrifice". Maybe it's coming. Maybe I haven't noticed it yet. There are certain video-game characters that we ritually slaughter. Certain large corporations we hate-on. (Is Microsoft our "devil"?) But I'm not sure we've got sacrificial gods yet.

See : http://www.quora.com/Philosophy/What-becomes-of-our-stigmata-martyrs-when-the-gods-we-create-and-destroy-are-no-longer-made-in-our-own-image-but-in-the-image-of-our-own-creations

Why do programmers tend to fall in love with non-mainstream languages?

It's not that we ONLY love non mainstream languages. I admire as beautiful, elegant and expressive (let's say "love" for the purposes of this question) C and Python too and they're totally mainstream.

But when a language isn't mainstream, "love" is the only reason to use and talk about it. Because there aren't other reasons (eg. I have to use this in my job or to use this platform)

See : http://www.quora.com/Computer-Programming/Why-do-programmers-tend-to-fall-in-love-with-non-mainstream-languages

Will writing my own OS put me into elite group of developers?

I'd give you props for it.

As others have said, the chances that it will take off and get users is pretty miniscule. But if that's your itch, go scratch it.

See : http://www.quora.com/Will-writing-my-own-OS-put-me-into-elite-group-of-developers

What is the most implemented programming concept?

“printing” the words “hello world”. I don’t think there’s a single programming language or dialect of programming language that doesn’t at least make a valiant attempt to do that.

See : http://www.quora.com/What-is-the-most-implemented-programming-concept

What are different programming levels of abstraction useful for?

Startups might work at lower levels of abstraction if they are

a) working on things which are closer to hardware. Eg. embedded systems, drones, robotics (http://www.quora.com/Robotics), 3D printing, home automation etc. etc.

b) creating novel infrastructure where someone else hasn’t already created the higher-level of abstraction (ie. that wheel hasn’t been invented yet)

See : http://www.quora.com/What-are-different-programming-levels-of-abstraction-useful-for

Why are “abstract” classes called such, when they do not represent abstractions?

Abstract Classes *are* abstract in the sense that they can’t be instantiated as concrete objects in your program. (Only subclasses of them, that fill in the missing definitions can be.)

See : http://www.quora.com/Object-Oriented-Programming/Why-are-abstract-classes-called-such-when-they-do-not-represent-abstractions

What are some of the most fun things you’ve ever programmed, and why?

I think the highest payback I’ve had in terms of long term fun / value relative to the amount of work it took is Gbloink!

I wrote the original over a weekend and although I adapted it over time I’ve spent far more time playing with / composing with it than the time I actually spent coding it.

This new version that runs in the browser took about 2 hours once someone had finally invented a suitable library I could use :

See : http://www.quora.com/Computer-Programming/What-are-some-of-the-most-fun-things-youve-ever-programmed-and-why

The Craft

How do you avoid premature abstraction?

Ian Bicking has a very good post on "". And I think my reply (interstar) in the comments is good too.

I'm not sure there are hard and fast rules. Identifying commonalities that are worth refactoring out is an art rather than a science. Smart people can legitimately disagree.

But there are things I'd say are worth paying attention to :

1) It's not just about *lines* of code. Instead try to get a sense of the number of decision points or "bits of information" that any piece of code represents. Even half a line of code can be worth abstracting / refactoring if it represents several decisions which would be easy to forget or be inconsistent about.

A good example is any time you find yourself constructing a tuple it's worth being alert to the possibility of refactoring. Eg. if you find yourself writing (name, age, weight) more than once, it might not be worth creating a special class but at least a constructor function to put such a tuple together. Why? Because despite being so short, that piece of code represents several *decisions* … what goes into the tuple and what order they're in. That's the kind of thing that's easy to forget. If I found myself writing this in three or four places I'd probably abstract.

A similar case is when you construct a string out of several parts. If it's important that that string is consistent, consider a special constructor for it.

On the other hand, if you have a huge block of boilerplate that doesn't really represent any decisions you've made, it may actually not be that urgent to abstract away from it. Though I would just to get rid of visual pollution.

2) Another way of thinking about this. Some bits of code are the same because they DO the same thing. But other bits of code are the same because they MEAN the same thing. The second are more urgent to abstract than the first.

3) But, of course, the value of abstraction also depends on how cheap it is to make them. A language where everything demands a class, is heavier than a language with free-floating functions.

4) Consider what namespace you're cluttering up with your new abstractions. If it's the global one, that's worse than cluttering up the module you're in. Even better is if you manage to contain your abstractions within the private members of a class or even as local functions within a single method.

5) Abstractions may or may not save you absolute numbers of characters in the short term. But they should ALWAYS make the code that uses them easier to understand. The name of your abstraction should always correspond to something you can fluently.

One of the problems for people learning Functional Programming is that the abstractions are so unfamiliar they don't seem to make the code easier to read because you have to go and look up what a foldl means. Obviously FP people know those terms intimiately and have no problems. But be aware of making abstractions with names that don't help you understand what the code using them actually does.

See : http://www.quora.com/How-do-you-avoid-premature-abstraction

How can we minimize logic errors and refactoring when coding, fast?

You absolutely DON'T want to minimize refactoring. Refactoring is like calisthenics. It keeps your code supple and in shape.

Ward Cunningham (one of the great gurus of this) has written that it's sometimes worth making "deliberate" mistakes to practice the art of making changes ( ). I believe that he's right. It's far better to be good at making changes than to think you're so good that you can write code which doesn't need changes made to it.

(Also :)

See : http://www.quora.com/Computer-Programmers/How-can-we-minimize-logic-errors-and-refactoring-when-coding-fast

Programming Languages

Tech Culture

Free Software

What is the most successful open source project so far, and why?

In terms of software :

- GCC
- Gnu/Linux
- Apache
- Firefox

in that order, but closely followed by a bunch of languages, libraries and several decent end-user packages.

BUT

the most successful project has been the invention of / definition of "free software" / "open source" / "open culture" itself.

The very idea that such a thing is possible and the right thing to do. Which has spawned a legacy in all the other projects, not only software but hardware from Arduino to RepRap to Open Village Construction Set to hundreds of crowdfunded projects all of which commit to giving away their software and design schematics.

It's hard to remember but in the 80s it was just kind of "obvious" that software was going to be (big) business and computing was going to be expensive. Even if there was going to be "home-computing" it would be hobbled by licenses that forced you to buy a full-price copy of the software for every computer it ran on. And by dongles and copy-protection etc. With serious home software packages costing several hundred dollars and even basic business software creeping up into the thousands, most people would only get to try and use a few packages.

Individuals might pirate, but that wasn't an option for legitimate companies. So let's remember that without Linux and other free-software tools, companies like Google and Facebook would have been born "in chains". Paying a per-server operating-system tax to Microsoft or Sun. And with Microsoft more or less determining what they could do. (Would Google be able to compete when Bing came out calling undocumented APIs in Windows Server?)

In this sense we have to add the shape of the web today to open-source's credits.

Pretty much every successful service (large or small) got that way using free-software to ensure its freedom from any particular platform provider. (BTW : the next generation of would-be-giants will have to relearn this lesson and win their freedom from Apple, Google, Facebook etc. through adopting free and open communication protocols / distribution channels etc.)

The *idea* of free / open-source software is FLOSS's most successful project. And the FSF's GNU General Public License (https://www.gnu.org/copyleft/gpl.html) was a crucially important part of spreading that idea. The GPL was a concrete artifact. As important in its own way as the American Constitution. It got people talking and thinking and arguing about software freedom and whether code should be shared or hoarded. Even when they rejected the license as too stingent, many people had to do so from an enriched understanding of what was at stake.

For some it was mere convenience. But many poeple adopted the GPL as a badge of pride : a banner to signal that they also stood for freedom and contributing to the commonly shared wealth of networks. Today it's hard to find serious software developers who don't rely on, and recognise the value of such a commons and feel that some aspect of it should be honoured and protected (even when they have particular business models that are in conflict with it.) That is an extraordinary change in mindset in the last 30 years and an extraordinary victory.

See : http://www.quora.com/Open-Source/What-is-the-most-successful-open-source-project-so-far-and-why

Is free software/service sustainable in the long run, other than a mechanism for corporations to collect personal info for ad revenue?

There are two different meanings of the term "*free software*".

There's software you don't pay for. This might well be crap because it's funded by advertising and the danger with that is that the real customers of the product are the people buying the advertising space. For this kind of software, the user will always be a second class citizen.

Increasingly frequently today, it may also be crap because it's a vehicle for "in game purchases". That is, the original software (usually a game) is a loss-leader with the hope of selling you a lot of upgrades, accesses to higher levels etc.

Although there's nothing wrong with this in principle (it's just the well established model called "fremium" where the basic level is free and the premium version is paid) it seems to be being done in a fairly exploitative way by some companies, who lure you in with a lot of flash and then try to get huge amounts of money out of you during the game, often relying on the psychological techniques that games use to get you "hooked". You care about your character and your progress and so you feel compelled to keep buying the upgrades.

There's another, completely different use of the term "free software" which is by the "free software community" which means that thing that is commonly also called "open source". This is software which is free because the people who write it believe in people's freedom to read, learn from, adapt and copy whatever information happens to be useful to them and these people want to contribute to an ecosystem of such free programs and tools.

This kind of free-software community falls into two rival camps : those who believe in freedom as an ideological good, and those who believe in freedom as a pragmatic good that leads to better software. Although they used to snipe at each other, members of both communities usually work together perfectly well in practice and have created a huge ecosystem of tools that are free. The Gnu/Linux platform (usually shortened to Linux) is a great example of this. It has millions of completely free, extremely high-quality programs available. And people keep contributing to this collection every day.

Becuase there's this problem of ambiguity in the term "free software". people in the movement sometimes distinguish between the terms "free as in beer" (to mean software that's merely unpaid, but which may exist because the developers have other ways they want to exploit you) and "free as in speech" which captures the political ideal of of "free software" which respects your freedoms. (See more here :)

It's in your own interest to try to understand this distinction and learn to recognise which type of "free software" you're dealing with. If something say's it's a "free download" but has no reference to links to let you see the source-code or discussion of the ideology of the free software movement, you should probably be suspicious. It's likely to be "free as in beer" and may have other agendas behind it.

If you see it being hosted on SourceForge or GitHub or other sites which are associated with the free software community. If it's on Linux. If you see links to the source-code (even if you have no idea what you would even do with that); that means its more likely to be a product of a community who are making it available because they want to share in the benefits of a free, commons-based ecosystem of tools. And it's likely to be fairly high quality. (Though sometimes it's a bit geekier and has a steeper learning curve.)

But like I say, for your own good, learn to distinguish the two. I use almost nothing but "free as in speech" software and it is *excellent*. I would rather pay money to buy proprietary software than let "free as in beer" software onto my machine.

See : http://www.quora.com/Is-free-software-service-sustainable-in-the-long-run-other-than-a-mechanism-for-corporations-to-collect-personal-info-for-ad-revenue

Why is open source hardware not as successful as open source software?

Being physical and scarce, there are inescapable constraints (in terms of cost, transport times etc) to hardware.

It will never have the same profile as software which is infinitely copyable using hardly any resources and which downloads over the internet in seconds.

Nevertheless, many open-source hardware projects have become successful (Arduino and RepRap are probably most prominent)

Popularity here means not only are new people continuously adopting and making the designs but are making variants on the designs which feed back into the ecosystem and the common stock of knowledge.

Perhaps some variant of the Raspberry Pi or other ARM based board might do too. Sooner or later we're going to have a fairly well established open designs for laptops and mobile devices. It will still be a challenge to find people to manufacture them, but I think the growing scene around open-source hardware might get its act together to do this.

One huge bottleneck is silicon. Chips require multi-billion dollar fabrication facilities. But maybe we'll get more processors emulated with generic FPGAs or generic combinations of FPGA and simple CPU cores.

See : http://www.quora.com/Why-is-open-source-hardware-not-as-successful-as-open-source-software

Is it OK to use icons with a GNU License in your commercial mobile application?

I'd *guess* (IANAL) it's OK if you also put the icons somewhere easy to download on your site and explicitly say that people can reuse them in their own applications.

See : http://www.quora.com/GPL/Is-it-OK-to-use-icons-with-a-GNU-License-in-your-commercial-mobile-application

Is it possible to compete with free? If so, how?

Look for customers who are either stupid (Linux vs. Microsoft) or suffer from low self-esteem (Linux vs. Macintosh).

See : http://www.quora.com/Marketing/Is-it-possible-to-compete-with-free-If-so-how

What can and can't I do with open source software licensed under LGPL?

No, you don't need to share your changes back if you aren't distributing UNLESS the code is licensed under GPL Affero ( ) where you must share it even if you just run it on a public server.

See : http://www.quora.com/What-can-and-cant-I-do-with-open-source-software-licensed-under-LGPL

Can software that modifies a library under GNU GPL be sold without releasing the proprietary source to the software?

"This implies that any tech company that uses modified GPL software would need to release their code if they charge for their service."

Forget the "if they charge for their service" bit. But *yes*.

If you use GPL software in your code-base, then you have to make your code available for others to use. However much you'd like to keep it proprietary you can't. That's the bargain you enter into with the Free Software community when you choose to use their GPLed code.

There are a couple of legal loopholes, but you shouldn't care about them, because the principle is what's important here. Everyone in tech. benefits when software is free-software. The GPL just exists to remind companies that they shouldn't try to abuse and screw up the cornucopia for a temporary individual advantage.

See : http://www.quora.com/Open-Source/Can-software-that-modifies-a-library-under-GNU-GPL-be-sold-without-releasing-the-proprietary-source-to-the-software

Static Typing

What is the strongest argument against statically typed programming languages?

Let's start with an example. The current state of my trying to get to grips with Haskell's Yesod framework : Serving CSS documents from Yesod (https://stackoverflow.com/questions/22801406/serving-css-documents-from-yesod/22804368)

Haskell is a good example because, unlike Java, it has type inference which allegedly removes a lot of the verbosity of static typing. And Yesod is a poster-child for the virtues of static / strong typing in a web-framework.

And, look, the guy who created Yesod, responsively listening to my problem and updating his framework for me. How awesomely cool is that?

It really is the best case scenario.

B..b..b..but …

You can't avoid the fact that if you build a language that defaults to saying "NO" instead of "YES" you will inconvenience people who are not making a "mistake" in the conventional sense. There's nothing wrong with me wanting to return CSS from my Yesod app. And there's nothing *prima facei* incorrect about wanting to map URLs to style-sheets in the parseRoutes I'm passing to mkYesod.

It's just that the author of Yesod hadn't thought of it that way, and so it was blocked because a strong type-system requires things to be actively whitelisted as "allowable". The fact that he immediately recognised it as allowable and fixed his framework to allow it, shows that it wasn't controversial in any sense.

But it required a special appeal to him. He became a "gatekeeper".

And note that he has added code especially to handle my case of wanting to return CSS.

The problem is that I have an ongoing project where I write software that generates STL files for 3D printers. Is it likely that the author of Yesod has foreseen my requirements for this too? And has added a type for STL? Or will I just duck out of the type-system at this point and return some kind of generic plain text or binary file?

There is an unresolvable tension here. In strong / static typed languages people who write the frameworks and libraries have to be able to look into the future and predict all the possible requirements that their users will have. Otherwise the type system will block those future uses by default. (Or people will take what opportunities there are to bypass the type system by reverting to working with generic strings etc.)

(Although I'm not overly fond of Libertarian politics, I think there's a good analogy here, with their distinction between top-down centralized planning associated with governments, and the bottom-up self-organization associated with markets. Centralized planning is never going to enable as much experimentation and progress as a distributed, market, system where everyone is free to pursue their own explorations.)

Stringent typing requires heroic oracular capacity from those who build the libraries and frameworks. Otherwise all the *accidental* constraints that are baked into those components and enforced by the type system will be a continuous yoke around the necks of users of those frameworks. That's the Java experience that we all know and hate : code bloating with little conversion functions and wrappers (https://stackoverflow.com/questions/9327695/yesod-how-do-i-serve-dynamically-generated-css/9340842#9340842) to push things through the type-guarded pipelines. And despite Haskell's wonders, it might well turn out to be the experience of the growing number of Haskell users struggling with a growing number of frameworks, each of which failed to predict just one or two of each user's particular needs.

Although static typing advocates often claim that their way allows better scalability than the weaker / dynamic typing, at the very largest scales of all, the protocols used by billions of people and applications on the internet, have to follow Postal's Law ("Be liberal in what you accept, conservative in what you transmit.") Societies can only flourish when tolerance for the unorthodox is wired into their DNA. And I believe the same is true of technological ecosystems.

See : http://www.quora.com/What-is-the-strongest-argument-against-statically-typed-programming-languages

Is static type checking overrated?

It's very hard to say. Advocates of static typing will tell you that the compiler is always picking their bugs up saving them time.

As someone who is usually comfortable in dynamically typed languages I tend not to interpret my bugs that way. I certainly make mistaks. But after I've spent an hour or two tracking one down I don't tend to think to myself "damn! if only I'd been using Java the type-system would have picked that one up". It may be the case that the type system *could* have picked up the bug but the code would have been so different that it's hard to know whether the cost / benefits would have worked out.

I can honestly say that I never *miss* static type checking when don't have it. Perhaps after I've spent a bit more time with Haskell (my language to get to grips with this year) and started to feel the benefits I may change my opinion.

See : http://www.quora.com/Software-Engineering/Is-static-type-checking-overrated

When languages don't require explicit data types to be provided, how do they infer a type?

From the values that are assigned to them.

Eg.

x=3

The virtual machine knows that the value 3 has the type "int" or "number" and so x also carries that type.

If you then say

x="hello world"

x takes the type from the value, a string.

Sometimes it's ambiguous. Is 3 an int or a floating point? The answer is that it could be either. In this case, the language will probably default to int and then convert to float if we try to use it in an expression with another float. This automatic type conversion CAN bite you, especially in languages that do it with ad-hoc, unprincipled rules (Javascript is notorious for this). But, most of the time, it's pretty straightforward and type-conversion gives you what you expect and want.

See : http://www.quora.com/Programming-Languages/When-languages-dont-require-explicit-data-types-to-be-provided-how-do-they-infer-a-type

Programming Languages and Teaching

How would you rank your favorite programming languages and why?

1) Clojure : My new infatuation. With all the Functional and Lispy goodness you've been hearing about. Very nice indeed. To be honest, it might have been another Lisp, but contingently it happens to be Clojure that I first started using in earnest. (And the JVM integration is important for how I'm using it) Read this entry as Lisp if you prefer.

2) Python : The workhorse. I mean, "it just works". Sure there are all sorts of languages that have better features. But, to repeat : Python "just works". You don't even have to think about writing it. For 90% of what I want to do it more or less writes itself.

3) CoffeeScript : Python that runs in the browser (where you want your code to run). And without some of Python's more egregious failings. Perhaps a nicer language overall. But not quite so convenient everywhere outside the browser. (This entry subsumes Javascript which you could call CoffeeScript with a clunkier syntax. CoffeeScript is also the language most "at risk" of being knocked off this list, if self-hosting ClojureScript ever takes off.)

4) C : It's actually a cleverly designed language for its purpose : to be close to the machine but portable from one environment to another. Its longevity and ubiquity is testament to how much better than most of its rivals it's actually been. In 2014 we can all imagine much better. And maybe one of those languages (eg. Rust / Go) will finally deliver.

5) Erlang : Nice FP with a great concurrency story. But I'm now finding I like Clojure's Lispiness more.

6) Smalltalk : Wonderful language. But sort of missed out on being where it mattered.

7) Haskell : "Look it's not you, it's me. You're amazing! Sexy, sleek, powerful. Mind-blowing. But at the end of the day, I find I just can't be with a demanding, bossy type-system." I do try it every now and then, but it just doesn't suit my style of thinking.

8) PHP : Yeah, everyone hates it. But serves a unique niche well.

9) Java : Bleah!

10) Cache ObjectScript (MUMPS) : I worked for a couple of years as a MUMPS programmer. Maybe one day the nightmares will ease off.

See : http://www.quora.com/Programming-Languages/How-would-you-rank-your-favorite-programming-languages-and-why

Which programming languages are in your "to study" queue?

Haskell is my current "get to grips with" language. What sold it to me was Tikhon Jelvis's point that you could use it to write DSLs that compiled to stand-alone C. And I really want to use properly as it becomes more mature. Looks very promising.

I have an eternal "do something with Lisp" in my "sometime" bucket. But I never quite get there. It almost happened this year. I was considering having a go at Clojure, particularly to see if it made writing Android apps. more palatable.

But it turns out what's faffy in Android seems to stay faffy in Clojure. In fact, Clojure's selling point is that it makes Java faff possible from Lisp rather than Clojure has good ways to hide it.

You can compile Pi-Occam for the Arduino now () and I'd like to have a play with that.

I've also played with Erlang but never written a reasonable sized application with a lot of interacting nodes. I'd like to try that.

In general my feeling is that (Functional) Reactive Programming is the new "garbage collection" ie. that just as what characterised the important mainstream languages from the 90s on was their hiding responsibility for memory management from the programmer, what will characterise the next big wave of improvements in mainstream languages is hiding responsibility for explicit event / call-back handling.

So any language that promises to implement FRP or even just reactive programming as conveniently as possible is something I'm interested in learning.

See : http://www.quora.com/Programming-Languages/Which-programming-languages-are-in-your-to-study-queue

There are many programming languages like C,C++, Java, Python and many more each having its own benefits.Which language you prefer and why? What are the benefits of using that language?

For low level stuff and embedded systems.

C : Because I know it and think it's surprising elegant when used well.

One day I may try Rust as a replacement. And on Arduino, I'd like to try Pi-Occam. Though apparently Go is also making a running here.

In the browser

CoffeeScript is my default. Though I'd like to try Elm. And as I'm playing with Haskell currently, I'm also looking into other Haskell to Javascript compilers for some libraries.

For other "application" development (everything from small-scripts, to desktop apps. with GUIs)

Python is still my first choice (familiarity, does what I need) but increasingly I'm writing a server and using the browser as my GUI, so writing the UI part in CoffeeScript. I can forsee a moment when I might move to node.js and an all CoffeeScript solution.

On Android

Java : I don't like Java and I wish I wasn't writing it. But for the small amount of Android programming I do I'm just using Java until someone can make a convenient higher-level language that engages the process easily.

For the Web

Python traditionally, but may move to Javascript / CoffeeScript again.

I'm intrigued by Erlang. If I had a good reason I'd try it more seriously. Also, if the whole Haskell things works out, I'd look into Haskell solutions.

The Future

Haskell : It's my "learn this year" language. So far I'm doing small experiments but I can see that I may end up using it in several situations : as command-line tool particularly for parsing and pre-processing other kinds of data, for various music apps, and to more elegantly write some of the more complex libraries that are compiled into javascript (and perhaps called from Elm or CoffeeScript in the browser)

See : http://www.quora.com/Programming-Languages/There-are-many-programming-languages-like-C-C++-Java-Python-and-many-more-each-having-its-own-benefits-Which-language-you-prefer-and-why-What-are-the-benefits-of-using-that-language

Is PHP a programming language or an overengineered template engine?

What's the difference? Really?

See : http://www.quora.com/Is-PHP-a-programming-language-or-an-overengineered-template-engine

What makes any one programming language better than another?

Let's try to keep this simple :

- Clean syntax
- Good abstractions
- Not too oppressive.

“Clean syntax” is fairly easy to understand. Don’t make people write a lot when a little will do perfectly well. The cleaner the syntax, the better, because it gets visual noise out of your face and lets you concentrate on the meaning and structure of the code.

“Good abstractions” is a huge area. But it boils down to one thing : can you express a lot with a little code. In general if a language allows you to do something in 2 lines instead of 4 then it’s a better language. With the proviso that you aren’t just squashing a lot of language constructs onto the same line.

Good abstractions let you put a lot onto one line because they are ways to express more general ideas. A function call is an abstraction, instead of saying “do all this stuff” each time, you define it once and just tell the computer every time you want to reuse it and with what parameters.

There are even more exotic abstractions than that, though. Comprehensions let you say “this transformation on all elements of that collection that satisfy this criteria”. There are abstractions that say “all things that are like this, will behave like that in those circumstances” and you have to do very little else. (Some type-systems allow you to express this. Abstract / mixin classes get you some way there too.)

“Not too oppressive” is the most subjective of these three criteria. Because, as with fathers, one person’s idea of “unfair oppression” is another’s idea of “useful discipline”. But given that possible subjective disagreement, languages that just allow you to do what you want / need to do are better than languages that put a lot of fussy rules and bureaucracy in your way, either because they just think it’s “for your own good” or because they’ve decided you’re always going to be writing nuclear power-station control systems where nothing must ever be allowed to go wrong. Even though you currently want to process a bunch of log files.

See : http://www.quora.com/What-makes-any-one-programming-language-better-than-another

What are the factors to be considered when building a new programming language?

There’s no limit to the human imagination. As in all the arts, there’s always something new to be discovered in programming language design.

We’ve no more “finished” inventing programming languages than we’ve “finished” painting pictures or composing music.

But having said that, Toby Thain (http://www.quora.com/Toby-Thain) is right that you should take time to consider some history before embarking on it.

We've all been there … the point when we decide our tools aren't good enough and that we'll revolutionize programming by essentially reinventing C / Java / Python with a weird syntax and a couple of our favourite idioms hardwired into the language.

If you only know a bit of C / Java / Python it's really worth looking at programming language history, at Functional Programming, at languages with concurrency and event-handlers built-in, at Prolog, at Dataflow languages, at Forth-like stack-based languages, at Smalltalk's self-contained world. You'll find that the space of possible programming languages is way larger than you could have dreamed.

If you want to dream big, make sure you know how big, big gets.

On the other hand, don't be intimidated. A punk musician with the right attitude can be as fresh and innovative as a classical composer. Just because some programming language designers are incredibly clever that doesn't mean that the right bunch of string processing scripts hacked together can't change the world. (That's how Perl and PHP started and both have earned their place in programming language history, by occupying their niches well and empowering millions of programmers.)

What's important is to make something different, usable and useful. Solve that problem and your language may become one of the greats.

See : http://www.quora.com/What-are-the-factors-to-be-considered-when-building-a-new-programming-language

If you could predict one language that will take over the programming industry (over C++, Java, Python, etc) which would it be? (Ruby, Perl, Haskell, Go, etc)

Javascript is going to be very widely available and used. It's already moved to the server with node.js. With node-webkit (https://github.com/rogerwang/node-webkit) it becomes a viable way to write cross-platform (between Windows / Mac / Linux) desktop apps. I think that will put it in strong competition with Java and the JVM on those operating systems.

It's already the "native" language of ChromeOS and Firefox OS. And with PhoneGap it's also one language for cross-platform mobile apps.

No one language will take over everything or everywhere but I think the Javascript VM has a good chance of becoming the most widely used and important platform to develop for. Which means that other languages will be rushing to compile for it.

That means, moving forward, less emphasis on things like Jython and Clojure and more on Brython and ClojureScript. Elm-lang, CoffeeScript and all the other compile to Javascript languages. To an extent, some of these languages will "leak" the semantics of Javascript. CoffeeScript is a good example as it's just Javascript with a nicer syntax. But I think we'll find compromises being made with some of these other languages too.

There'll probably be an increasing effort to compile Java and its libraries to Javascript.

The Javascript VM may not be ideal for this, but I'd guess that the makers of things like V8 can steal good ideas from the JVM and CLR just as JVM borrowed from Strongtalk etc.

See : http://www.quora.com/If-you-could-predict-one-language-that-will-take-over-the-programming-industry-over-C++-Java-Python-etc-which-would-it-be-Ruby-Perl-Haskell-Go-etc

Which is the most powerful programming language of all time?

It's still early days for "all time" :-)

Lisp is a good contender but not necessarily for the reasons that a naive Paul Graham fan might give you.

You can say Lisp is the most powerful language. But that's sort of like saying "strings of characters" are the most powerful language. Yes, you can say anything you like with strings, but only when interpreted with the right semantics.

Although Lisp brings some semantics - the nested lists do represent a kind of abstract syntax tree - it's a very generic structure that can be wrapped around almost any semantic interpretation. And over time, the semantics of Lisps have changed. From dynamic scoping to lexical; the introduction of macros and hygiene; lazy evaluation; immutable data etc.

In practice, today's contenders are probably Clojure, which combines many of the advances of Lisp with access to all the Java libraries. And Haskell which does all kinds of magic with powerful types and monads.

See : http://www.quora.com/Topmandu/Which-is-the-most-powerful-programming-language-of-all-time

What are the most useful and practical programming languages to learn?

There is no programming language that can force itself to run on a computer that isn't set up for it.

So you have three options :

- use a language that compiles to machine-code binaries for the kind of computer you want to run on. Maybe you're thinking of Windows which is pretty popular. But remember if you compile for Windows your program still won't run on Mac, iPad or Android tablet etc.
- use a language which is widely supported. For example, the Java virtual machine is already installed on many desktop PCs (Windown, Mac and Linux) and servers. But not all.

- accept that users will have to make some effort to install your program on their machine, which might include installing the language.

Javascript is possibly the programming laguage that runs across most platforms, including desktops and smart-phones, by virtue of the fact it runs in the browser. However this comes with a HUGE caveat. The browser security model doesn't let javascript code do things like read and write files from the disk or access other secure resources.

So if you want to write a program that lets people do certain calculations, Javascript might well be what you're looking for. But if you want to process data stored in files it is absolutely NOT the easy cross-platform language you' might think. It can run cross-platform if the users install something like node.js, but this an equivalent extra difficulty to asking them to install the Java Virtual Machine or Python.

See : http://www.quora.com/Computer-Programming/What-are-the-most-useful-and-practical-programming-languages-to-learn

What is a list of programming languages ordered from easiest to hardest to learn?

Learning the syntax and basic philosophy is different from learning to use properly is different from learning to use well.

That's at least three different orderings right there.

See : http://www.quora.com/Programming-Languages/What-is-a-list-of-programming-languages-ordered-from-easiest-to-hardest-to-learn

Which programming language has the easiest syntax to learn?

Syntax isn't really the hard part of programming languages.

See Phil Jones' answer to Computer Programming: What are some of the most common misconceptions/myths about programming? (http://www.quora.com/Computer-Programming/What-are-some-of-the-most-common-misconceptions-myths-about-programming/answer/Phil-Jones)

Why is this important? Well, Lisp has a very simple, easy to learn syntax. But it's still pretty difficult to understand how to use. Syntactic simplicity has very little to do with any simplicity you might care about.

See : http://www.quora.com/Which-programming-language-has-the-easiest-syntax-to-learn

What would be a good first programming language?

Between Ruby and Python, toss a coin. There's just not enough difference to get worked up about. Python probably has more online courses and tutorials. But Ruby has some well thought of ones too.

See : http://www.quora.com/What-would-be-a-good-first-programming-language

Which programming languages are most "fun" to use and why?

Python is pretty fun because, for people with general imperative / OO backgrounds, it gets out of the way and just does what you want. That's always pleasurable.

When I first starting writing Javascript (coming from working with mainly C++ and Perl) I found that it had similar characteristics. Just doing what I wanted without fuss or weirdness.

Functional languages give you a certain kind of buzz when you suddenly see how you can do quite complex operations on data structures with very short recursive code. You're suddenly, like, "wow! that actually does everything I need. I took half an hour to figure it out but now it's 4 lines of code. WTF?"

The original Visual Basic. Back in the days when it was all about Windows and GUIs and that stuff was complicated, VB was a breath of fresh air. You just dragged a couple of components into place, double clicked and wrote event handlers and "job done" in a couple hours. Compared to the alternatives at the time, or even to building web-based GUIs today, it was surprisingly low stress.

See : http://www.quora.com/Programming-Languages/Which-programming-languages-are-most-fun-to-use-and-why

Why do the most productive programming languages (e.g. Haskell, Scala, Clojure) remain in obscurity while less productive and unmaintainable ones (Java, VB, C++) enjoy mainstream dominance?

You misunderstand "productive". The languages you mention are objectively powerful, but there's more to productivity than power. There's also availability and convenience.

Nothing was more productive than VB to knock up a simple form-filling / database accessing Windows program back when such things mattered. Nothing is easier than PHP when you want to throw up a simple form-filling / database accessing web-page on cheap web-hosting. Nothing is easier to knock off than a quick shell / Perl / Python script to munge a file at short notice.

There's a Zipf's law (https://en.wikipedia.org/wiki/Zipf%27s_law) of software design. MOST of the programs that get written are these small, ephemeral ones. And only a few are giant 50+KLOC monsters that need long-term maintainence. So languages that are good for quick and dirty convenience cover far more of the programs than the languages that have features that only become important at scale.

Today, we're finally getting to the stage where some powerful functional programming languages are ALSO becoming convenient. (They have fast compilers, good libraries and tooling support). Don't underestimate how recent this is though. Clojure is a Lisp that runs on the JVM and accesses a wealth of Java libraries. But for the previous 20 years Lisps have been either expensive or lacking in library support. Haskell may be suitable for GUI development today, but I doubt it was comparable to VB in the 90s.

See : http://www.quora.com/Programming-Languages/Why-do-the-most-productive-programming-languages-e-g-Haskell-Scala-Clojure-remain-in-obscurity-while-less-productive-and-unmaintainable-ones-Java-VB-C++-enjoy-mainstream-dominance

What's the difference between a programming language and a scripting language?

There isn't a hard and fast difference.

Scripting languages are typically languages that are designed to quickly tell some kind of system / platform what to do. As opposed to write large-scale software. But it's not deep and meaningful distinction.

Typically :

- they're interpretted by some kind of virtual machine, rather than compiled.

BUT C has been called "the scripting language of unix".

And, in practice both Java and Python are compiled to code that runs on a virtual machine. Java just makes this step explicit and Python doesn't. Python is considered a scripting language while Java isn't.

- they're used for small-scale programs that do one thing.

BUT today people are writing increasingly large applications that run the browser using javascript. And there are some huge Python / Perl / Ruby programs.

- they're "higher level" (ie. have run-time rather than compile-time binding of things like variable names to types)

But really, the difference is not a deep and formal classification. It gets an idea across quickly, but the distinction is fuzzy.

See : http://www.quora.com/Programming-Languages/Whats-the-difference-between-a-programming-language-and-a-scripting-language

How do scripting languages improve functionality?

“Scripting languages” is a bit of a vague term but it usually covers languages that :

- come with an easily / transparently available virtual machine / interpreter, so there’s no distinct compilation stage. And therefore no having to think about compilation. Nor having to debug compilation options (ie. linking external libraries) etc. before you see anything running. Sometimes, half-way through running you find you’re trying to include or reference something that’s not there. But you don’t have that miserable phase of trying to pin down all kinds of obscure problems before you get the psychological lift from seeing your program half-run. (Which is common in say C++ where you can’t figure out what library you should have linked or where it is.)
- are usually dynamically typed, so you don’t have to spend time up-front declaring types or making sure your type-constraints in different parts of the program are aligned. This means you may bump into some errors later that compile-time type-checking would have caught. But in practice, when you’re writing short scripts, or writing your scripts incrementally, you don’t hit as many of these as you might fear.

As these make clear, part of the advantages of scripting languages is psychological. They let you get into a rhythm of write something, get a bit of stimulation from seeing it run, feel energised to write the next thing. Languages with more explicit compilation or static types hold you back earlier, preventing you make these mistakes, but also preventing you get into this “groove” (ie. into the Flow (http://en.wikipedia.org/wiki/Flow_(psychology))-state).

See : http://www.quora.com/How-do-scripting-languages-improve-functionality

Why wouldn’t you want to teach an 8-9 year old child assembly language for their first programming language?

8-9 year olds want to program to achieve RESULTS. They’re pretty focussed in that sense; and they largely want to make games (I know I did, when I started programming (at around 11).)

Learning assembly is not the most direct-way to get results. You have to learn deep machine architecture (which might have been a necessary to get anything done in the past, but isn’t now).

And more importantly, where will the code run? How easy is it for the child to share what he / she has produced with friends?

OTOH, teach them to write javascript in the browser and you can throw the result onto a site so that it's a click away from any social network the child might be using.

Caveat : there is ONE exception to this. Maybe you're teaching your child electronics with something like an Arduino. In which case, thinking about machine architecture IS still important and maybe assembly is more suitable. (It is still harder than C, but maybe that's not such a big jump.)

See : http://www.quora.com/Why-wouldnt-you-want-to-teach-an-8-9-year-old-child-assembly-language-for-their-first-programming-language

Is Ruby on Rails a good choice as a first programming tool for 13-15 year olds? If not, what alternatives are there?

Distinguish between Ruby and Ruby on Rails.

Rails is designed to make certain kinds of database backed web-site easier. Unless the 13-15 year old wants to create that kind of site, RoR isn't going to help much.

Even if they are, I'd be inclined to start with the whole "what's inside a web-page" thing : a bit of HTML, bit of CSS, bit of javascript. Learn how to do simple games / applications inside a single web-page.

Then, when they've got a grasp of programming (loops, if statements, functions, objects) from that, move out of the browser to more general languages that control other things.

See : http://www.quora.com/Programming-Languages/Is-Ruby-on-Rails-a-good-choice-as-a-first-programming-tool-for-13-15-year-olds-If-not-what-alternatives-are-there

Is there a lot of overlap in programming knowledge?

```
       For all values of n ... your n+2th language is easier to learn than your\
n+1th language. Except where two languages are trivially similar.
```

What are the top five languages every programmer should learn?

Javascript ... because everywhere C ... because Von Neuman architecture Lisp ... because maths Haskell ... because hipsterism[1] No, sorry, I meant ... because more maths. Python ... because need to eat / get shit done / not Java.

Strictly speaking, language-wise you could substitute CoffeeScript for Javascript AND Python. But it would be kind of cheating on the spirit of things here, because you won't really learn CoffeeScript without learning Javascript anyway, OTOH it *would* free up a slot for

Prolog … because … that feeling in the series finale when just when you wrapped up 80% of the problems from earlier episodes, some weird shit happens and you realize there's waaay more to this than you thought and next year's series is going to be even more of a headfuck.

[1] Circa 2014, 5 years ago it would be Ruby.

See : http://www.quora.com/What-are-the-top-five-languages-every-programmer-should-learn

What are the three most important programming languages to learn?

normally people ask about 5

But if it's 3 it's easier : C, Lisp, whatever you like.

See : http://www.quora.com/What-are-the-three-most-important-programming-languages-to-learn

If I could learn one language between Scala, Python, and C++, which one should I choose and why?

None of the above.

C++ is just C with clunky OO attached. If you want to get into that kind of mind-set, learn Java instead which is a) easier (no worrying about memory maagement), b) has a wider range of applications.

Python is a great language but will teach you very little you don't already know from writing Javascript. (Use CoffeeScript as a syntactic sugaring for Javascript and it's almost identical)

Scala's main attraction is that it's a functional programming language that's on the Java ecosystem. That means a lot of its ideas only make sense BECAUSE you're trying to engage Java from FP. If you want to learn FP ideas, learn Haskell or Lisp because Haskell is the most advanced and exotic FP language out there (of the ones you're likely to want to learn) and Lisp is the quintessential FP language. (You can also learn Clojure if you want Lisp on the Java ecosystem.)

See : http://www.quora.com/If-I-could-learn-one-language-between-Scala-Python-and-C++-which-one-should-I-choose-and-why

What do you think of C programming language?

C is a gem. It's one of the best (certainly in the top 3) programming languages of all time.

People forget that. Because it's so everyday. And because we've all had absolute hell working with it. (Segmentation Fault, Core Dumped anyone?)

But C itself is a fantastic bit of programming language design : simple, concise, incredibly powerful (if you know how to use pointers to functions and the void* type, you can emulate many of the late-bound / higher-level virtues of object-orientation and functional programming.)

Perfect for its original purpose of allowing people to write code which was portable from one machine / operating system to another (Forget Java. C is the original "write once, run anywhere" language. All you have to do to port is to set some flags and run the compiler for the new architecture.)

It's no accident that C is everywhere. It beat out the competition time and time and time again. The ultimate convenient and pragmatic choice.

It's only now, when machines are around 6 orders of magnitude(!) more powerful than when C was invented, that we start to think of higher level languages like Python or Javascript or Scala as viable competitors.

Basically you should and will learn C if you want to understand most areas of software development. And certainly if you want to understand programming language design.

See : http://www.quora.com/C-programming-language/What-do-you-think-of-C-programming-language

Is learning C++ still worthwhile?

It depends. I wouldn't try to learn it as a first language. C++ is what we call a "low level" language. One which makes you think about the system you're programming rather than the algorithms.

To learn programming I'd go for something higher-level. (In the old days we all used to use BASIC to learn programming. These days we use Python.)

BUT as you're doing an electrical / electronic engineering course you will indeed have to think about the low-level / hardware oriented style of programming. And for that C/C++ is essential. It may be old but it's still the basis of most software systems today.

See : http://www.quora.com/C++-programming-language/Is-learning-C++-still-worthwhile

Why do so many people said say it takes years to learn C well?

It takes years to get good in any programming paradigm. It's superficially easy to blunder about and get stuff done, but recognising when you're doing things "well", when you're working with the grain of the language and not making life hard for yourself by fighting, it these take a lot of experience.

C is an absolutely fantastic language because it looks so superficially low-level and simple. There are a few basic ideas that combine well. But when you get good, particularly when you have experience of thinking at higher-levels, you find its expressivity scales.

One of the most profoundly enlightening things I ever did, happened by accident when I was teaching a basic undergrad comparative programming language course. I was giving a taster of Lisp to the students, trying to get them to think recursively about some problem or other. And as a way of making it clearer to those who might be confused by the alien Lisp syntax I did a direct translation of the tree building / searching algorithm into C.

I expected it would be horrible. Or at least highly verbose. But what came out was a surprisingly short and elegant. Although C isn't Lisp, it can concisely implement a few equivalents to the Lisp fundamentals which then compose very well.

I understood Greenspun's 10th law (that all large C programs have a buggy imperfect implementation of Lisp inside them) and realized that the way to take advantage of that is to embrace it, to build the list-processing primitives and implement the data-structures and Lisp-like algorithms using them.

Unlike C++ where you are lured into thinking in terms of classes and building verbose, inflexible models of the world, raw C is surprisingly good at letting you approximate the styles of higher-level functional languages. Function pointers let you pass functions around. Structs of function pointers become Objects and Classes. I believe, now, that good C programmers can turn C into the language they want it to be.

But such understanding requires a lot of experience. Both in C and with other languages.

See : http://www.quora.com/Why-do-so-many-people-said-say-it-takes-years-to-learn-C-well

Is C language more prone to bugs based on the experience of HeartBleed?

C is more prone to bugs than higher level languages. Yes.

Not due to Heartbleed. But perhaps Heartbleed has made more people aware of the fact.

The bad argument FOR C, is that it needs this low level access for performance reasons, and that there's an inevitable trade-off where it can't have safety without the unjustifiable performance hit.

It's a bad argument because computers are pretty fast nowadays and so there few cases where this raw low level performance is really necessary. Moreover, languages can be a lot cleverer than C about how and where they allow risks, so that you could, in principle, have languages with 99% of the performance with safety 99% of the time.

The *good* argument for C is that, despite the complaints of its critics, it's a very good language. A great pragmatic mix of elegance, low-level access, power, expressiveness. Whereas today we tend to think of "C-like" languages getting a boost from the laziness of programmers who don't want to learn a new syntax, C itself took around 15 years to take over the world. And did it on its own merits. (OK, maybe the rise of Unix helped somewhat, but Unix wasn't growing as fast as the PC which could have brought a different language to prominence in the 80s if a genuinely better one had been available.)

C dominated because none of the rivals was actually as good in all the dimensions they needed to be. They may have had virtues that C doesn't, but they didn't satisfice as well.

Maybe today we have the experience and a large audience with a taste for new languages that someone will be able to invent a C killer. Maybe it's D, maybe it's Rust. Maybe it will turn up on Hacker News tomorrow.

It will probably need several things :

- a compiler to C itself, to ensure it can run more or less everywhere C does.
- really simple calling into C dynamic-libraries
- easily called from C
- good integration with the tool chain. gcc support would be very valuable indeed. (The aim is to make writing modules in the new-language transparent to the development process.)

See : http://www.quora.com/Is-C-language-more-prone-to-bugs-based-on-the-experience-of-HeartBleed

What are the novel ideas and profound insights in the design of the C programming language?

The really big idea in C was hardware independence. C was a language designed to be compiled and run on different hardware and even operating systems.

Not necessarily the first, but certainly one of the most successful.

Beyond that, it's a very well selected / fine-tuned collection of good pragmatic ideas that can actually give a lot of power to the programmer, while giving a lot of access to the low-level machine (byte-level control of memory) and control to write efficient code.

Kernigan's critique of Pascal gives some interesting insights of why C is the way it is :

See : http://www.quora.com/C-programming-language/What-are-the-novel-ideas-and-profound-insights-in-the-design-of-the-C-programming-language

What are the novel ideas and profound insights in the design of the Python programming language?

Python wasn't the first language to use whitespace / indentation, but it brought the idea into the mainstream. One of Python's real successes was convincing people that they could break out from C-like syntaxes.

Everyone other popular language in the 90s (Java, Javascript, Perl, PHP, C# etc.) more or less assumed that languages had to look like C.

In a sense, Python was a bit like the Apple Macintosh of programming languages of the time : embodying the belief that design / style / appearance was as important as semantics. Things "just work" rather than promote an ideology. Doing things the *right* way is easy. Doing things the *wrong* way is almost impossible.

See : http://www.quora.com/Python-programming-language-1/What-are-the-novel-ideas-and-profound-insights-in-the-design-of-the-Python-programming-language

How powerful is Python as a programming language?

Pretty powerful by most people's standards.

There are things you can do in Haskell or some varieties of Lisp that are probably more mind-blowing. In my personal experience, I can write things in Erlang that are about a quarter of number of lines that I would need to write in Python.

But you can go a long way towards understanding and using fairly high-level "functional style" programming in Python.

And unlike these more exotic languages, Python is *very easy* to get into and work with. Has a huge standard library and other popular, easily available libraries and frameworks for what you want to do. And it will certainly give you a productivity boost compared to things like Java / C# / C++ / PHP etc.

See : http://www.quora.com/Python-programming-language-1/How-powerful-is-Python-as-a-programming-language

Why are interpreted languages (e.g. Python) popular in security?

I don't suppose they are, particularly. It's probably more the other way around.

Interpreted languages are popular. So there's a demand for people who know how to make them secure.

See : http://www.quora.com/Why-are-interpreted-languages-e-g-Python-popular-in-security

How does one create a Python web application?

1) Decide if you really want to.

Why? Because while Python is a wonderful language, its most popular web-frameworks : Django, Google App Engine etc. are focused on the old-style of page-based interactions, wrapped around relational or quasi-relational databases.

It's clear that the new trends in web-design focus on the page being a rich dynamic application in its own right, sending tiny fragments of json data backwards and forwards, often to synchronize an in-browser data-model with a backend, NoSQL model.

The direction is clearly signalled in frameworks like etc.

You can certainly do such things in Python, but I suspect that the main energy in Python web-frameworks (and similarly Ruby on Rails) isn't on supporting that. And maybe the language isn't quite as amenable to it as server-side event-driven javascript in node.js or some functional languages that provide the equivalent of continuations.

2) If you're really a beginner and just want to learn the basics, try which is probably the simplest framework to get started with.

See : http://www.quora.com/Python-programming-language-1/How-does-one-create-a-Python-web-application

What's a good web dev platform if I work in Python, but want something faster to set up than Django?

I use for small scale Python web-apps. It's minimal but does what I want. Not sure about the email bit, but may do.

See : http://www.quora.com/Python-programming-language-1/Whats-a-good-web-dev-platform-if-I-work-in-Python-but-want-something-faster-to-set-up-than-Django

Why is it so difficult to set up a programming platform to learn Python or Ruby?

Eclipse is particularly egregiously hassle. It's very weird to think that Eclipse should be considered the default Python solution. (Even though Eclipse is so popular that for many Java-turned-Python programmers it *is* their default.)

I personally used to like Wing for Python many years ago. And today I just use a plain editor. (gedit in Linux, notepad++ in Windows) and that's fine for my purposes. I've never tried PyCharm but it seems to be popular. And ActiveState are still going with their products (eg.)

See : http://www.quora.com/Computer-Programming/Why-is-it-so-difficult-to-set-up-a-programming-platform-to-learn-Python-or-Ruby

I am experienced java programmer but new to python. How should i set up my development environment in an industry standard way? Note: I am using windows 7.

1) Follow Pramod Lakshman (http://www.quora.com/Pramod-Lakshman)'s advice.

2) On Windows, a decent editor / IDE is probably a good idea. IDLE is OK to play with, but you probably want something better. Either one of the good general editors for Windows or something like .

3) I highly recommend source-code management. Git is becoming standard these days, so a version of Git is useful. Particularly if you can set it up to automatically convert Windows to Unix line-endings. (If you do any web-server type work with python, even when developing on Windows you'll probably be deploying on Unix)

See : http://www.quora.com/Python-programming-language-1/I-am-experienced-java-programmer-but-new-to-python-How-should-i-set-up-my-development-environment-in-an-industry-standard-way-Note-I-am-using-windows-7

What are the kinds of applications that are not suitable to be developed using Python?

The big crisis facing Python is the rise of Javascript.

Bluntly, much of the UI of applications has migrated to the browser. And with things like meteor.js you're starting to see code which is written to span the browser and server. Not just using the same language at both ends, but defining functions and data-structures which migrate transparently between the two.

node.js is increasingly popular serverside. node-webkit lets you write full desktop GUI apps. in javascript / html5 / css.

Python is a far nicer language than Javascript, but CoffeeScript is pretty close to Python for most purposes.

So there's interesting work to make Python compile to Javascript (eg.), so that like CoffeeScript and ClojureScript etc. it can be a full browser citizen. but it's hardly the first choice for in-browser apps.

Similarly, it's not really making much of an impact on mobile development, where Java and Objective-C still dominate and there doesn't seem to be much of an opening for Python-based development.

See : http://www.quora.com/Python-programming-language-1/What-are-the-kinds-of-applications-that-are-not-suitable-to-be-developed-using-Python

Why is JavaScript getting more popular, especially for server-side code?

For most programmers, I think, the syntactic / idiomatic differences between say, Javascript and Python/Ruby/Perl/PHP etc. are trivial. It's not really the language being the same that counts.

In fact we've had javascript (via Java) available on the server for ages and no-one cared much.

What seems to be the big win is to have the javascript event / callback semantics for free on the server without having to use some kind of weird library for it.

People are used to event-driven programming in Javascript. Node.js gives them that server-side in a lightweight, fast form. Plus Coffeescript gets rid of most of what you superficially DON'T like about javascript.

So I'd say the reason for the recent explosion in popularity is a) node (fast and lightweight), b) CoffeeScript (nicer syntax, classes), c) people used to / liking the javascript event model.

See : http://www.quora.com/JavaScript/Why-is-JavaScript-getting-more-popular-especially-for-server-side-code

What's the future of Javascript?

There's no end of Javascript currently in sight.

It's the native language of the most important platform / virtual machine that exists today (inside the web-browser).

On that platform it has access to the two most important client-side graphics libraries today : HTML5 as a GUI widget-set and OpenGL for (hardware accelerated) 2D and 3D. Through HTML5 it also gets websockets for networking and a slew of other capabilities.

It's has several popular solutions for running server-side; solutions for desktop GUIs (node-webkit) and even mobiles (PhoneGap).

As Mattias Petter Johansson (http://www.quora.com/Mattias-Petter-Johansson) points out, it's a high-level language that allows, even encourages, "functional programming" style which is clearly the direction that much programming is going in the future.

Even those who don't like Javascript are increasingly looking at the Javascript VM as a target for compiling other languages. CoffeeScript gets rid of the ugly and verbose C-like syntax and makes Python and Ruby programmers happy. There are compilers for everything from C to Haskell via Java and Python to the Javascript platform (although they don't always have the libraries or access to system level resources you'd expect). And there are languages like ClojureScript and Elm-lang which give you variants on Lisp and Haskell that are optimised for browser-scripting.

So the foreseeable future is very much Javascript + whatever higher-level language you might adopt to compile down to it (many of which, like CoffeeScript, allow some of the semantics of Javascript to leak through.)

See : http://www.quora.com/JavaScript-programming-language/Whats-the-future-of-Javascript

How easy is it to learn Java if already know C++ and Python?

Your C++ experience will help a lot more than your Python experience.

When you think of it from a C++ perspective, it's generally simpler (Java is like C++ without having to manage memory de-allocation and with a different templating system).

From Python you'll be continuously disappointed and frustrated with all the things that aren't there : modules, functions as first-class citizens and higher-order functions, comprehensions, generators, decorators, multiple-inheritance etc. and with all the extra fussiness of the type-system.

See : http://www.quora.com/How-easy-is-it-to-learn-Java-if-already-know-C++-and-Python

Is coding Java in Notepad++ and compiling with command prompt good for learning Java?

It won't be a *pleasurable* experience, because, let's face it, the premise implies

a) that you're in Windows.

and

b) you'll be writing Java without an IDE.

Java is horribly verbose AND finicky, and more or less designed on the assumption that you're using an IDE do to the grunt work for you. (I'm not a big fan of Eclipse at all … it's slow and clunky. OTOH I don't think I could write Java without it. I'm a mediocre Emacs user at best, and my experience of writing Java in Emacs was pretty unpleasant too.)

However, it probably *will* be an educational experience.

And if you *have* to use Windows and don't want to use an IDE, notepad++ seems a pretty good editor. I use it for writing Python and Haskell in Windows and it's OK. I mean, it's not like trying to use Windows' own notepad etc.

I'd still way rather work in Linux though.

See : http://www.quora.com/Is-coding-Java-in-Notepad++-and-compiling-with-command-prompt-good-for-learning-Java

What are the novel ideas and profound insights in the design of the Erlang programming language?

From the History of page :

That seems a good encapsulation of Erlang's *unique selling point* when it was conceived.

So Erlang was one of the languages designed to have concurrency and parallelism baked in. While drawing a lot on both Prolog (pattern matching and syntax) and Lisp (functional programming).

It standardized (though didn't invent) the Actor model.

See : http://www.quora.com/Erlang/What-are-the-novel-ideas-and-profound-insights-in-the-design-of-the-Erlang-programming-language

Why does Erlang have far more real world usage than Haskell in terms of IO centric services?

Erlang is that rare thing : a programming language invented by industry for its own internal use.

Most languages come from academia or research projects and are based on what the researchers think will be a good idea. Or are intended as teaching languages to try to inculcate the right values into students.

Languages that come from industry OTOH have tended to come from platform vendors for whom the language is either a cost of doing business. ("Oh God! I suppose we need a Fortran on our OS for all those engineers") Or is designed to promote the platform. ("Look VisualBasic makes writing Windows programs easy").

In contrast, Erlang's motivation is seen as purely practical. It has been dog-fooded by Ericson since its invention. And its sales pitch is great : "Sure it's weird. Sure it looks like Prolog! (WTF????) But here's a bunch of rock-solid switches that it helped us make." That's something that immediately attracts attention and commands respect.

See : http://www.quora.com/Distributed-Systems/Why-does-Erlang-have-far-more-real-world-usage-than-Haskell-in-terms-of-IO-centric-services

Which language is most commonly used to build Linux apps?

Linux is much more pluralistic than these other platforms. Many languages are used.

C was the original language of Unix, so a lot of Unixen (including) Linux have a lot of code written in C.

C++ has been used for a lot of Linux software too.

The Mono project ports Microsoft .NET to Linux, and some code is written in C#.

Some Java apps. which are cross-platform are also popular on Linux.

Python is increasingly popular as a scripting language in Linux. It can be used to write GUI apps. via GTK or wxWindows etc.

Perl is the more traditional scripting language, not so much for GUIs, but for command line tools.

TCL was once a popular scripting language and you'll find apps written in that too.

Javascript has moved out of the browser with node.js. That means that there are an increasing number of apps. for Linux that have a server written in javascript and use the browser as the UI.

Some people like to write in Ruby, Haskell, Scheme, Guile, Pascal etc. etc.

See : http://www.quora.com/Which-language-is-most-commonly-used-to-build-Linux-apps

What type of programming language is used to create iCloud and Dropbox, all big sync back and restore tools?

I believe Dropbox is Python

update: https://tech.dropbox.com/2014/04… (https://tech.dropbox.com/2014/04/introducing-pyston-an-upcoming-jit-based-python-implementation/)

See : http://www.quora.com/What-type-of-programming-language-is-used-to-create-iCloud-and-Dropbox-all-big-sync-back-and-restore-tools

How did Apple's programming language Swift get its name?

You want a snappy, one-word name. Like "Go".

You want to suggest something easy, that just flows.

You want an animal, because, as O'Reilly have conclusively demonstrated, all programmers are closet shamans at heart.

You've already used big cats for your operating system.

Genius steals.

See : http://www.quora.com/Apple-Swift-programming-language/How-did-Apples-programming-language-Swift-get-its-name

Is there something in dead languages you wish would have transferred to modern programming? Did we miss any great ideas?

There are a lot of ideas that keep coming back.

Dataflow was a category of programming language that never hit the mainstream, but had ideas that are coming into fashion now as "reactive programming" or "functional reactive programming".

Prolog's inference engine is available as libraries in other languages. And it would be interesting to see if this kind of inference could be made more accessible.

APL's special characters for powerful matrix and other mathematical operators may make a comeback. If you can have *domain specific languages*, why not domain specific *syntax*? Especially as we move away from using standardized keyboards towards tablets with touch screens and virtual keyboards.

I think we're going to see another round of high-level languages that orchestrate and script "swarms" of multiple computers. For clusters of servers, for service oriented architectures, and even for the increasing number of computers around your person (laptop, tablet, phone, watch). These may owe something to earlier service orchestration or business process modelling / dataflow languages.

Lisp never goes out of style. And has never really been a *dead* language. But you might have written it off as a historical / cult thing a few years ago.

However it's back with a vengeance as Clojure. And Racket is gaining popularity. In these two flavours, it's possible that Lisp will become more mainstream than it's ever been bringing the ideas of homoiconicity and proper macros.

See : http://www.quora.com/Is-there-something-in-dead-languages-you-wish-would-have-transferred-to-modern-programming-Did-we-miss-any-great-ideas

What are some of the dying programming languages in the next decade?

OK. Just to stir things a bit, I wonder if Ruby has peaked :-P

Ruby's burst of fame was almost entirely based on Rails. But current trends don't seem to favour it :

- the rise of rich HTML5 apps. that do more of their work in the browser in javascript.
- the rise of serverside javascript via node.js etc.
- the trend towards more dynamic queue-based message-passing services like WhatsApp which encourage developers to play with Erlang

- the general rise of functional programming as the new hotness (Clojure / ClojureScript, Haskell / Elm-Lang, Scala, Erlang etc.)
- the persistence of PHP as a reliable workhorse for a lot of more pedestrian web-work

Could Ruby go the same way as Perl?

Python suffers from similar trends, of course, but unlike Ruby seems to have some other strong power-bases : as a system scripting language, as a beginners' learn-to-program language (Python is the new BASIC), in scientific and academic computing. I'm not sure if Ruby has any equivalent areas where it's the preferred choice.

See : http://www.quora.com/Programming-Languages/What-are-some-of-the-dying-programming-languages-in-the-next-decade

Until RoR came along, was MVC popular? If not, what was the way to go back then?

I've always hated the MVC religion in web-apps. MVC was derived from patterns for desktop GUIs in Smalltalk back in the 80s. Applying it to the web, where the layers got conflated with the distinction between browser / server / database was always a misunderstanding.

There was certainly a lot of talk of it before RoR (for example, Java Struts (shudder)).

It seems things are improving now, though, as more code moves into the browser and we get a proper MVC distinction WITHIN the browser, and it's recognized that this is a distinct issue from the synchronization of data between browser and back-end.

See : http://www.quora.com/Until-RoR-came-along-was-MVC-popular-If-not-what-was-the-way-to-go-back-then

What design patterns from the GoF (Gang of Four) are outdated?

They're not so much outdated as specific to a particular kind of language with particular kinds of problems. (Mainly C++ and Java.) Some of the most famous patterns are work-arounds for Java's limitations. For example, why should you prefer reuse by composition (eg. strategy pattern) rather than inheriting from a superclass?

Well, because in Java you can dynamically swap out strategies at run-time but you can't dynamically add and remove mixin superclasses at runtime. But if your language allowed dynamically adding and removing mixins, then that would be equivalent to strategy pattern.

Or again, singleton pattern is just a way of faking module-level variables and functions. You don't have modules in Java so you make classes that pretend to be modules and have to make a deliberate effort to stop people multiply instantiating them.

See : http://www.quora.com/Design-Patterns/What-design-patterns-from-the-GoF-Gang-of-Four-are-outdated

How are abstract data types implemented in Java?

You declare some methods without method-bodies. Just the signatures, like you do in an Interface.

Or maybe I'm misunderstanding the question.

See : http://www.quora.com/How-are-abstract-data-types-implemented-in-Java

Why are some objects in some programming languages immutable?

If you have immutability it means that the only values in a function are constants and those passed in as parameters.

That, in turn, means that every time you call the function with the same parameters it will produce that same output.

Which in turn means that :

a) it's easier to unit test the functions. (Apart from the arguments, there's no extra state you have to worry about setting-up to have the right values)

b) it's easier to prove that the function does what it's meant to.

c) you can cache the results of calling functions on particular argument sets for speed. (Memoization)

d) if there's no "state" ie. extra information in your functions beyond your parameters, then there's certainly no "shared state" which means it's much easier for a compiler or virtual machine to distribute your program across multiple processors or machines.

The more your program is organized around functions and immutable data-structures like this, the more amenable it is to these benefits.

Obviously, some FP languages (Haskell) pretty much force this on you. Some like Clojure strongly encourage it. And many support it.

See : http://www.quora.com/Why-are-some-objects-in-some-programming-languages-immutable

What programming languages can return more than one value from a function?

Most popular languages allow you to return a tuple or list.

And some languages have "destructuring assignment" of the kind you're using in Go. (Eg. Python, CoffeeScript have it.)

Lower-level languages like C can allow a function to fill an array which is passed by reference. Or to dynamically construct some kind of container on the heap and return a pointer to that. But don't have destructuring assignment.

See : http://www.quora.com/Programming-Languages/What-programming-languages-can-return-more-than-one-value-from-a-function

Can I rename open source libraries licensed under LGPL?

I'm pretty sure yes. As long as you make your changes available for other people to download if they want to.

The GPL doesn't want to put any constraints on you apart from preventing you restricting the redistribution of your changes, so I can't see that it would put extra constraints on naming.

Of course, obfusticating the naming scheme might be an excellent way to discourage anyone actually taking your changes and merging them back into the main code-base ;-)

See : http://www.quora.com/Open-Source/Can-I-rename-open-source-libraries-licensed-under-LGPL

Industry and Technology Trends

General Trends

What are some successful contrarian trends in software technology or technical culture?

A “successful” “contrarian” trend might be an oxymoron.

But here’s something that struck me yesterday when reading which is a very good essay that seems to signal the direction that many smart people think software development should be evolving in : namely giving up on as much explicit state and control flow as possible and moving towards a declarative style or saying just what your program should produce without worrying about how it does it.

I can’t overemphasize how big this idea is. Most important and smart people thinking about software will sign up to the idea that we need to move towards more functional languages, more declarative style, abandon more state and explicit control flow. Perhaps even separate the essential logic of what you want done from the “accidental” hints that can enhance performance into separate languages / parts of the system.

And yet …

And yet, the most widely adopted, commonly used example of this separation of telling what the program should do in one language and performance hints in another, (acknowledged in the paper) was the good old fashioned relational database written using SQL; which did, indeed, allow programmers to declare what they wanted their queries to deliver without worrying about *access paths*, control flow or performance. And then database admins worked behind the scenes profiling, creating special indexes etc. to improve performance.

Now, since this important paper was written, there’s been an absolute revolution in database circles, called the NoSQL movement, a wholesale rejection of the relational database model and its replacement by systems that hark back to the hierarchical and network databases of the late 1960s, Although NoSQL was adopted by people working on *enormous* systems across hundreds of thousands of machines, its popularity is so great that a new generation of programmers reaches for NoSQL database solutions (and explicit modelling of data-structures and responsibility for traversing access-paths etc.) more or less by default, even for small prototypes.

So, I’d say that NoSQL is one of the most successful “contrarian” movement. It’s massively popular and “trendy” while going against everything that many smart programmers think and say they want, and what many people had forseen as the future of software development.

It signals either that the argument in Out of the Tar-Pit is wrong : namely because performance is so important that programmers never want to give up explicitly modelling state and defining control-flow, or that people’s intuitions are badly broken.

See : http://www.quora.com/What-are-some-successful-contrarian-trends-in-software-technology-or-technical-culture

Will "knowledge-based programming" languages like Wolfram Alpha be superior to more traditional languages like Java?

More likely "knowledge-based programming" will form a certain kind of niche.

Wolfram Language will certainly help to expand that niche and the number of people analysing certain data-sets with certain methods. And that could make the world a much better place.

It won't replace a lot of the programming that's needed for the underlying plumbing of the system. And probably won't replace the experimental programming that's needed to invent new kinds of applications either.

The "English-like" nature of writing queries should probably be quietly ignored. Such "natural language" programming languages don't have a good history (see Phil Jones' answer to Computer Programming: What are some of the most common misconceptions/myths about programming? (http://www.quora.com/Computer-Programming/What-are-some-of-the-most-common-misconceptions-myths-about-programming/answer/Phil-Jones) for more details). And I expect the natural language abilities of WL to fail for the same reason : once you start wanting to phrase complex questions in the form of very specific transformations of data, English turns out to be extremely verbose and imprecise.

That's why we invented mathematical notations in the first place : to be able express complex and abstract ideas in an precise and unambiguous way. Even when talking to each other.

So WL will certainly have to have a less verbose / more "traditional" syntax, basically Mathematica. If you're worried that writing in Maths notation is somehow going to make *everyone* a programmer in a way which Python doesn't, then calm yourself. Your job is safe.

Ultimately the success or failure of WL (or the "knowledge programming" paradigm) depends on how much knowledge the creators / maintainers of the language put into it. Is there enough? Is it up-to-date? Is it sufficiently queriable / navigable?

Wolfram is trying to do the same thing that the Semantic Web people did - which, personally, I think is, at best, VERY HARD, and at worst, IMPOSSIBLE - namely predict the kinds of data-structures people wil need BEFORE those people actually decide what applications they want to write.

But Wolfram *is* very smart. And he is rich enough to try. And maybe he'll discover a sweet-spot where he has enough valuable data in the right format to appeal to a sufficiently large customer-base to make this viable. It's a heavy commitment though (to keep those databases up-to-date). I'm half expecting Google to (at least try to) buy him out. In the long run a project like this needs the scale of data capture / management that Google have.

See : http://www.quora.com/Will-knowledge-based-programming-languages-like-Wolfram-Alpha-be-superior-to-more-traditional-languages-like-Java

Are there any flow-based graphical programming tools like Yahoo pipes, but with any local piece of software?

Great question. I can't say I know anything very close to what you're asking.

There are a lot of dataflow systems like Pipes which are specialized for music or video production. Max/MSP, PureData, VVVV etc. are like this. It *is* sort of possible to do other things with them too, though they aren't convenient for it.

Apparently Facebook released a Quartz Composer plugin for web development : You can now build an interactive mobile app, no code required, thanks to Facebook (http://venturebeat.com/2013/12/20/you-can-now-build-an-interactive-mobile-app-no-code-required-thanks-to-facebook/)

What might be interesting to look into is whether someone has produced a graphical dataflow front-end for one of the build systems (see List of build automation software (https://en.wikipedia.org/wiki/List_-of_build_automation_software)) Haven't looked into it myself, but would be fascinating to know. And may be the easiest approach.

See : http://www.quora.com/Are-there-any-flow-based-graphical-programming-tools-like-Yahoo-pipes-but-with-any-local-piece-of-software

Was object-oriented programming a failure?

If it's a failure I dread to think what success looks like.

Less gnomically, by any realistic measurement it has been a great success. Most new languages have some kind of objects. Many people find objects AND classes useful to organize their code. I know I do in some circumstances.

When people say it's failed, what they mean is that it didn't turn out to be the panacea silver-bullet that some of its boosters hoped it would be. Or that the Java marketers promised it would be. That's fine, we don't believe in panaceas and silver bullets. We know programming is hard. We know large scale maintenance is harder. We know the world will always catch out our best attempts to predict next year's requirements and make us sad.

Now the FP boosters are smugly thinking to themselves .. "yeah yeah … but if Java had never taken off and everyone had been using CommonLisp or Haskell or OCAML since the early 90s things would have been so much better".

Bollocks!

I think FP is wonderful too. I like what I've seen and done with it so far. It's a "good thing"(tm). But FP is still relatively untested in the wild. Don't try to persuade me that if the same clowns (bored and uninspired cubical dwellers in stodgy enterprises, hundreds of fresh-out-of-college kids with more attitude than wisdom) were armed with FP they wouldn't have made just as much of a hash of things as the Java generation did.

Right now, the FP community is self-selected from of the smartest and most discerning people in computer science. But FP is rapidly getting fashionable. And sooner or later it's going to be the thing that you have to pretend to buy into in order to get a job. And the same people that you currently castigate for not understanding MVC and for putting business logic into the JSP templates will be putting business logic into the IO Monad and using macros and DSLs to not only fake their favourite imperative styles but invent dozens of incomprehensibly weird pet idioms that mungle up core logic with UI expediency and work-arounds for the bits of the language they don't understand. (Boy, won't that be fun!) And then we'll be talking about "Was FP a failure?" too.

See : http://www.quora.com/Object-Oriented-Programming/Was-object-oriented-programming-a-failure

Are visual programming languages just for kids or learning purposes only?

Visual languages tend to fall into two types :

- ones that are aimed at teaching programming, for people who find words too dense and abstract. These are the ones that tend to feature "jigsaw piece" shapes to help people fit things together syntactically. And these are the ones you're probably thinking of as being for kids or novices.[1]
- ones that define dataflow networks. There are a tonne of these for sound and video (Max/MSP, PureData, VVVV), for simulation like Justin Rising (http://www.quora.com/Justin-Rising) points out with Simulink. And for process modelling.

The problem with using the first kind for real work is that all the extra graphical hints that help you see how to fit statements and fragments of statements together are very redundant and low density. You can't read a lot of them at once, and they're cumbersome to manipulate by dragging and dropping with a mouse, compared to typing on a keyboard. As soon as you learn what statements mean and how to fit them together, you no longer want to spend 70% of your screen real-estate on having the computer reminding you of that.

It also seems that such languages tend to be fairly standard procedural languages. You end up with the feel of a 1960s flow-chart diagram but little sense of function composition or the sort of class relationships you can represent with the UML.

The problem with the second group of visual languages is that while certain things are easy to express in them, other algorithms that DON'T fit the data-flow pattern well are fiendishly complicated.

---

[1]Anyone who thinks teaching kids to program should be about this absurd level of dumbing down should go back and look at what Alan Kay was able to get kids to do in the 70s. Alan Kay: Doing with Images Makes Symbols Pt 1 (https://archive.org/details/AlanKeyD1987)

See : http://www.quora.com/Computer-Programming/Are-visual-programming-languages-just-for-kids-or-learning-purposes-only

Internet Giants

Am I the only one who sees a big evil plan behind Apple's huge phones and their quite affordable new Apple Watch?

If you mean, are they planning to make them highly interdependent so that you can only use them properly and get the full benefits by owning and carrying both, then probably … a little bit … but not at the risk of making either so useless on its own that people won't buy it.

Apple certainly get that we're moving to the "device swarm". Computing that's exploded across multiple specialized devices that are hooked together through multiple wireless protocols. And they have their plans for it : http://www.tmcnet.com/usubmit/20… (http://www.tmcnet.com/usubmit/2014/06/03/7861835.htm

Now, on the whole, tech. companies split into two classes : those who want to own an ecosystem, however small, and hope to grow by growing it. And those who are willing to claim a piece of an existing ecosystem, and work with other companies within it, on the grounds that a small piece of a big pie is better than all of a smaller one.

Apple, with their full-stack philosophy are definitely in the first category. And their instinct is to want to control everything and exclude everyone else.

The "evil" of this is that they tend to reject / denigrate the open standards that allow users to connect their products to products from other suppliers. In the name of offering a "perfect" experience they will control everything from who can write software for your device to who can change the battery. And charge everyone a premium for the privilege.

If they can get away with it, they'll prevent your iPhone communicating as slickly as it could with Android Watches and vice versa.

BUT … they may not get away with it. Certainly not before the iWatch is fully established. So for the meantime they'll have to make sure that the products *do* work well together.

See : http://www.quora.com/Am-I-the-only-one-who-sees-a-big-evil-plan-behind-Apples-huge-phones-and-their-quite-affordable-new-Apple-Watch

Technical Specifics

Are big companies using PHP bothered by the fact that a lot of people say that PHP is a bad programming language?

Probably not.

PHP is a “trade-off” language. It has many ugly flaws, but it has some advantages, that other languages still can’t match. And it’s likely that any large company that’s heavily dependent on PHP, grew to be big, partly because of those advantages.

And, if you’re really a big company you can do like Facebook did, and basically rewrite a lot of the infrastructure behind PHP to make it a better, more secure and suitable language for you.

Longer term, all large technologically literate companies will be looking at better / cooler languages. They’ll continue to use PHP for the front-end (keeping their investment in PHP templates) but push off more back-end processing to higher performance, more secure, less bug-encouraging languages.

The other trend is that with the rise of HTML5 and faster Javascript VMs, more and more of the work of dynamically composing web-pages happens on the client and is written in Javascript. So I’d guess that PHP’s niche for assembling pages dynamically on the server is probably shrinking. I’d wouldn’t be much surprised if any large, mature system, originally written in PHP, is seeing PHP diminishing, as more of it migrates to client-side.

So any large company needs to be aware of trends and opportunities. But I wouldn’t panic. PHP has a lot of organic shrinkage ahead of it, without someone needing to make a dramatic decision to kill / rewrite it.

See : http://www.quora.com/Are-big-companies-using-PHP-bothered-by-the-fact-that-a-lot-of-people-say-that-PHP-is-a-bad-programming-language

What are PHP’s strengths?

- Widespread availability. (Plenty of cheap hosting accounts have it as standard) So if you just want to trial something without making a big commitment in terms of running a server it might fit your purpose.
- Lots of people know it. And programmers who don’t know it can get up to speed pretty quickly because of a C-like syntax (and Perl folk memories)

- Lots of web-designers know it, too, so it's easy enough to find a designer/developer who can work with it.
- Lots of tutorials online, aimed at designers (so not relying on existing programming knowledge)

See : http://www.quora.com/What-are-PHPs-strengths

What are the reasons for the death of COBOL as a programming language?

Basically there are lots of things that programmers today like and are used to, that COBOL doesn't have. Such as functions, block-scoping rules, dynamic memory allocation, garbage collection etc.

COBOL didn't used to have these, wasn't designed for them, and if they can be grafted on, it's probably a clunky misfit. (This is different from LISP which is of a similar age, but has a rather timeless design, which is so minimal that you can graft almost anything that's worth having on to it and it keeps its essense.)

Most of the things that were attractive features of COBOL were very much tied to the specific hardware, applications and times and are no longer in demand.

COBOL isn't going anywhere. It's fast, reliable and there are a hell of a lot of legacy systems that no-one dares replace that are written in it. But it's not a language which holds any attraction for people starting new projects.

See : http://www.quora.com/What-are-the-reasons-for-the-death-of-COBOL-as-a-programming-language

Is Java the language Cobol of today?

Yes.

See : http://www.quora.com/Computer-Programming/Is-Java-the-language-Cobol-of-today

Should Java stop promoting as a "write once, run anywhere" programming language?

It was always a stupid marketing slogan. Back in the 90s I was asking how many mouse buttons a run anywhere program expected. One (Mac), two (Windows) or three (Sun)?

See : http://www.quora.com/Should-Java-stop-promoting-as-a-write-once-run-anywhere-programming-language

I don't want to do Java any more. What should I do?

Obviously you can learn new languages by yourself, making personal projects. Make something that's interesting (or potentially lucrative) for you, not just something you hope will impress potential employers.

Age is absolutely not a problem for getting into new languages or ways of thinking.

"Pays well" might be more of an issue. Programmers are normally paid well either because of a) seniority in an established corporate environment, b) being part of a successful startup.

By definition you can't leave your corporate job / world and continue to have the money that you were being paid (partly as bribe) to stay in it. And you probably won't get paid so well while a newbie in a cool startup unless you bring something to them other than your newbie skills in their language. Maybe you find yourself become an "architecture specialist" for some startup which needs to scale up and rethink their architecture. Though be aware that many of the patterns you learned in corporate Java may be catastrophically irrelevant () If you have good intuitions about architecture, though, and make sure you're up on contemporary ideas (NoSQL, using local browser storage etc.) you might have a chance.

The other thing, Java is a language for doing cool stuff … in Android. You may find there's a role in a startup with people who've been doing web (PHP / Javascript) now need to up their Java skills to write apps. You get to be a Java expert / mentor, while simultaneously figuring out how to do things like use Scala / Clojure to make Android programming more productive.

I have no idea of these strategies could work out in practice, but might be worth thinking about.

See : http://www.quora.com/Computer-Programming/I-dont-want-to-do-Java-any-more-What-should-I-do

What are some tips to enjoy Java if you are used to dynamic languages?

Learn the Java design patterns (http://en.wikipedia.org/wiki/Software_design_pattern)

Patterns have got themselves a bad reputation these days, because like all good ideas (http://www.quora.com/Ideas), the more widely they're adopted and talked about, the more the original understanding gets diluted and the more people start to use them as a magic formula, and hence badly.

But the basic idea of patterns is perfectly sound. They're just a way for people who've figured stuff out, to document it, so that other people don't have to go through the pain of rediscovering it the hard way.

And a lot of the Java patterns actually exist to overcome the annoyances and inflexibilities of Java's static typing. Unfortunately they aren't always explained like that. And you won't see it immediately. But that's what they do.

The problem with static typing is NOT that it's boring. Or verbose. Those are minor annoyances. The problem with static typing is when you have an object being passed from module A to module B to module C to module D etc. and suddenly you discover that D is actually going to need a different type of thing. But now you've got to update not only A, but B and C, to change the types they're passing through. Statically typed programs freeze up and become inflexible quickly.

When you realise this, you start to notice that many of the classic patterns are about keeping your program as flexible as possible : Factories allow you to delay committing to exactly what class of object you want until as late as possible. Strategies allow you to swap-in new behaviours to already compiled classes. Facades allow you to pass new types of object through pipelines that were designed for older ones. Etc. Highly abstract interfaces and generics are the *mechanisms* that enable this, but it's the patterns that teach you how to apply these mechanisms to good effect.

Frankly, nothing is going to make Java as pleasurable as other, nicer, less-bureaucratic languages. But learning (http://www.quora.com/Learning) patterns will at least improve your "fluency" so that you write more supple, flexible code and so spend less time when you (inevitably) come to make changes to it.

See : http://www.quora.com/What-are-some-tips-to-enjoy-Java-if-you-are-used-to-dynamic-languages

Why don't more people use Python 3.x?

Because 2.x is still the out-of-the-box default on most Linux systems. Partly because the installed tools may (or may not) break if 3.x became the default.

Until 3.x becomes the default, most people will write 2.x by default. And the more people that write 2.x by default, the riskier it is to change to 3.x … I think there's a bit of a vicious circle there.

See : http://www.quora.com/Why-dont-more-people-use-Python-3-x

Is Python doomed if the Python community sticks to the old 2.x version, while the language continues to change with its 3.x version? Why?

I think it's a challenge.

It's not the biggest challenge, but I think, in retrospect, the idea that "we want to change some things, so let's change them all at once" might have been a mistake.

There is the uncomfortable example of Perl, which has never made the jump to Perl 6. When you make a big commitment to changing the language in a way that breaks backwards compatibility, people are inevitably going to decide whether they'll learn your language anew or whether they might choose to jump to a new language they're hearing good things about.

I kind of did this with some applications I'd written in VB6. I'd used VB6 in preference to Python because it was just easier. But when faced with the choice of learning VB.NET or making the effort to jump to Python (a language I preferred) I went with Python.

See : http://www.quora.com/Is-Python-doomed-if-the-Python-community-sticks-to-the-old-2-x-version-while-the-language-continues-to-change-with-its-3-x-version-Why

Will Python suffer the same fate as Perl?

Not yet.

I think it turns out that Python 3 was a bad move strategically. But it's not the disaster that Perl 6 was because it noticably "exists". Whereas Perl 6 was vapourware for a long time. And Python 2.7 and 3.x continue to develop similar libraries in parallel.

Worse still for Perl 6. Its first implementation was written in Haskell, which got Perl programmers *thinking about* Haskell. After which there were fewer Perl programmers.

So I don't think that Python programmers are going to fall out through the gap between 2.x and 3.x.

Still, it's a regrettable confusion. I suspect Python will continue with people recognising that it comes in two different "dialects" much as people accepted that there were different dialects of BASIC. And eventually one will just quietly die.

See : http://www.quora.com/Will-Python-suffer-the-same-fate-as-Perl

What are some examples of good programming languages that have failed to catch on and just faded away?

Arguably Smalltalk is the language that invented the GUI and the object-oriented style of programming; both of which took over the world for 20 years or so. And yet Smalltalk mysteriously failed to become as popular as it deserved, or as the things it spawned.

Instead C++, Java, Delphi, Visual Basic, Python and Ruby all went on to become far more popular, mainly using the tricks they copied from Smalltalk.

Why it didn't go mainstream is hotly debated and probably due to a variety of reasons. But I'd say the biggest was its (perceived) refusal to have anything to do with the file-system or other operating system features, and its stand-offish isolation in its own world of virtual machine / image.

OTOH Smalltalk is, still, a cult-classic, so perhaps doesn't quite fit your criteria of fading away.

See : http://www.quora.com/Programming-Languages/What-are-some-examples-of-good-programming-languages-that-have-failed-to-catch-on-and-just-faded-away

Are Smalltalk and Pharo out-dated?

Smalltalk is two things. A language and an environment (including standard library, a set of patterns of how to do thing, a virtual machine with its own storage system, assumptions about the kind interface people will use etc.)

Languages don't really date. And good languages are timeless. Lisp is the classic example of language that never seems to grow old. Though it does evolve a great deal.

Smalltalk is a very good language. Inspired by Lisp, and with many similar virtues. There's no reason that Smalltalk should go out of date.

Except … Ruby.

Ruby is a modern and very popular language which is sufficiently like Smalltalk that many (not all, but many) people who would otherwise be hankering after Smalltalk can be satisfied working in it.

Python is not nearly as like Smalltalk, but it was the first language that I picked up and said "OK. Now I don't have to keep fantasizing that maybe I can go back to Smalltalk." It had enough of the good stuff. Smalltalk is every bit as good as Ruby or Python (and written over 20 years before them!), but it's not obviously so much better that it can overcome the other obstacles to Pythonistas and Rubyists switching.

Now, the other side of the equation is the environment. Smalltalk was created in parallel with, and explicitly for, programming GUIs based on windows, icons, mice and pointers. And the irony is that, while this kind of GUI took over the world, a series of mis-steps in Smalltalk relegated it to a bit-player.

And now the era of the desktop machine and windows is drawing to an end. The important environments are the server, the browser and the mobile "device swarm" (soon to fragment into tablets, watches, glasses, drones and other robots etc.) While Smalltalk can certainly be made to work with these environments, it's not clear that there's enough need or momentum for it to become the "best" way to write for any of them.

Smalltalk is my first love. And I'm sad to say this. But I think it's going to be the great "also ran" of programming languages. Something that was extremely talented and influential but never achieved the popularity it deserved.

I'd love to see a tablet built around Smalltalk. The philosophy of the self-contained virtual machine, built-in library would work well. If someone could just do the work of making a slick multi-touch UI library and set of patterns. I'm not sure if anyone is.

But with things like FirefoxOS coming out it's clear that the browser - another self-contained virtual machine with it's own scripting language, and (with HTML5) own storage - is a direct rival (and far, far better known and understood.)

(See also Phil Jones' answer to Why did the Smalltalk programming language fail to become a popular language? (http://www.quora.com/Why-did-the-Smalltalk-programming-language-fail-to-become-a-popular-language/answer/Phil-Jones))

See : http://www.quora.com/Smalltalk-programming-language/Are-Smalltalk-and-Pharo-out-dated

Why didn't Modula-2 succeed?

Because Pascal sucks ..

No … sorry … mainly because Object Orientation took over the world then. And Modules weren't as cool as objects.

Objects are a good fit the the GUI Windows systems that were in the ascendant in the 80s. C++ retrofitted objects into C, which was the most popular language in Unix and making its way onto PCs.

Perhaps if Borland had made a Turbo Modula 2 things would have been different …

But at the end of the day … really, you say "easy syntax" but Pascal syntax does suck. Way too much visual noise with all those capitalised keywords. Ugh!

See : http://www.quora.com/Why-didnt-Modula-2-succeed

On a scale of 1 to 10 how dead would you say Perl is? And why should I stop learning and choose a language like Python or PHP?

I'm going to give it a 3 (assuming 1 is dead, 10 is HOT!)

That represents legacy status. I'm sure there are people maintaining useful Perl code-bases. And perhaps those people are extending, doing new work in Perl too. But I don't imagine there's anyone out there today who's doing a survey of languages in order to decide which one to adopt and thinking "Perl comes out on top".

It's not Perl's fault. It's a reasonable language. It's not as ugly as haters make out. (Really there's no reason your Perl should be any less easy to maintain than your C, PHP or Javascript.) But there's way too much competition in its space now from languages which are similar enough but have obvious advantages : PHP is more ubiquitous, Python has nicer syntax, Ruby has a massively popular web-framework. And Javascript will soon have all three of these advantages.

There are things Perl does better than any of these rivals. But it's hard to think of anything it does better that you NEED so much that it trumps their advantages.

See : http://www.quora.com/On-a-scale-of-1-to-10-how-dead-would-you-say-Perl-is-And-why-should-I-stop-learning-and-choose-a-language-like-Python-or-PHP

Which programming language got easier syntax in time but developed to become more powerful?

Perl

eg. https://en.wikipedia.org/wiki/Pe… (https://en.wikipedia.org/wiki/Perl_6#Syntactic_simplification)

See : http://www.quora.com/Which-programming-language-got-easier-syntax-in-time-but-developed-to-become-more-powerful

I keep hearing that some old languages like smalltalk and lisp are so great that new languages are still borrowing ideas from them, is that true? if that is true, why did these old languages die in the first place?

Largely lack of engagement with the underlying platform and ecosystem.

Smalltalk was very much a world to itself, except for a couple of expensive slightly non standard versions. Lisp didn't have access to the standard DLLs of Unix and windows or enough alternative libraries of its own.

In the 80s, compared to compiled C etc. these languages seemed genuinely slow and wasteful of processor resources.

Weird syntax is probably third compared the first two.

See : http://www.quora.com/Programming-Languages/I-keep-hearing-that-some-old-languages-like-smalltalk-and-lisp-are-so-great-that-new-languages-are-still-borrowing-ideas-from-them-is-that-true-if-that-is-true-why-did-these-old-languages-die-in-the-first-place

Why do programmers love Python and hate Visual Basic?

I don't.

I mean, I love Python, and I don't subscribe to the assumption that "all languages are equal".

But I think VB is (or rather, used to be) a pretty good language for the niche it was intended to fill :

- write simple Windows programs quickly (at a time when Windows was new and complicated and the only alternative was C / C++)
- design the GUI graphically (a rarity at the time, but something we all loved from HyperCard)

- use BASIC, the language that pretty much everyone knew, because they learned it in school in the 1980s on Apple, Commodore, Acorn / BBC, Sinclair etc. 8-bit home computers.

Today BASIC is unfamiliar because C-like languages, and then Python / Ruby became popular. Especially Python for teaching. So "looking like BASIC" now looks utterly old-fashioned, clunky and verbose. But it was a perfectly sensible syntactic sugar for the 90s. And it was much easier for casual programming than either C or Pascal derived languages.

Another thought I wrote up about 10 years ago.

See : http://www.quora.com/Why-do-programmers-love-Python-and-hate-Visual-Basic

Why will people decide to use TCL for projects in 2014?

Well TCL's selling point *used to be* the same as Lua's. It was a small, fast embeddable language that could be used for scripting inside other programs.

In the late 90s, early 2000s it became famous because it was embedded inside the open-source AOL Server, which meant you could do fast and dynamic web-sites with it without the overhead of Apache forking separate Perl or similar processes.

It also got a boost from Phil Greenspun creating one of the early comprehensive frameworks, the Ars Digita Community System, which was kind of the Rails of its day. (Well before Rails, and before Ruby, Python or even PHP got popular.) ACS's only rival was Slashdot's Perl-based Slashcode, which was a lot slower to run.

The other reason it was famous was tk. Which was the only way you could do light-weight GUI development in a scripting language on Unix, rather than writing C or C++ to call X11.

In 2014, I'm not sure if any of those are still relevant. I assume there are still some legacy tk or AOLServer systems which are being developed and extended, but I can't think of a reason you'd choose either of these frameworks / containers for new-build. Lua (or Javascript) are the embeddable languages of choice these days. Web-frameworks have evolved a long way. There's very little GUI development anywhere except apps. for mobiles. (Bastions of Java and Objective-C)

See : http://www.quora.com/Why-will-people-decide-to-use-TCL-for-projects-in-2014

Which computer programming language would be best to learn for the future?

looks to be a nice way to try out Haskell-like syntax and ideas, while getting Functional Reactive Programming, in an environment - the browser - which can be both fun to play in and very practical.

See : http://www.quora.com/Computer-Programming/Which-computer-programming-language-would-be-best-to-learn-for-the-future

How do companies make money by creating new programming languages?

Well, Microsoft got started by making a BASIC interpretter. But that's really another example of making tools, not languages.

In fact, there might be a good reason you can't make money by inventing languages. Success for languages requires them to be as widely used as possible . Restricting its use to paying customers will doom it. Anyone remember Rebol?

See : http://www.quora.com/Programming-Languages/How-do-companies-make-money-by-creating-new-programming-languages

What are the relative merits of closed vs. open source for a new JavaScript framework?

What do you actually want from making this "framework"?

1) If you want a community of many people using it in *their* sites, then it pretty much needs to be open-sourced because otherwise why would anyone choose to adopt and become dependent on it (and on you)?

2) If you just want to use it to build your own app. it doesn't really matter one way or the other. If you don't want to open source it, very few people are going to waste their time trying to grab it from your site because a) they'll have other alternatives that go out of their way to make it easy for them to adopt, b) people like to feel there's someone backing up their use of the technology they use.

3) If you are just going to use it in custom apps. for clients, see 2. But be aware that many clients will want the right to modify the framework as their needs evolve in future. If you don't discuss it, they'll assume they've bought the right to do what they like with the code. (And to bring other developers to work on it.) If you aren't giving them that right, you'll have to make a case as to why they should forgo it.

4) If you think you're going to sell it as a tool to other developers, you *might* have some success with that. But remember a) there are relatively few companies making a business from it. b) unless you have something spectacularly original and hard to copy, an open-source version probably exists or will spring up sooner or later.

There is a market for things like WordPress templates but these are usually sold to fairly naive users who wouldn't be able to find and adapt free alternatives for themselves.

Perhaps there's also a market for proprietary libraries (accompanied by a support contract), to sell to internal developers in enterprises which don't like or understand open-source. There your problem is to figure out how to sell enterprises. Good luck.

See : http://www.quora.com/What-are-the-relative-merits-of-closed-vs-open-source-for-a-new-JavaScript-framework

If Ruby is such a fun programming language to work with, why are there not a lot more web frameworks in Ruby?

```
	Rails got traction early. And was the reason why many people got into Ru\
by.
```

So a lot of Ruby programmers are de facto Rails programmers.

The situation is different in Python where people got into it for a wider variety of reasons, then realized they could / should use it for the web, and went off to create their web-frameworks. ## Are C# and the .Net framework suitable to create a modern OS like Windows 7?

This is a followup question from [the now deleted] thread: http://web.archive.org/web/20100812071342/http://stacko
windows-7-isnt-written-in-c

What kind of problems might one expect to encounter by choosing C# over the tried-and-true strategy of using C? What kind of problems in this domain might be better solved by a high-level language like C# compared to C?

A lot of Microsoft's problems in the last 10 years, with Vista and even Windows 7 seem to have been from trying to move to "managed code" of the C# variety. I'm not sure if the problem is the VM isn't robust enough or the GC is too slow. But it's not a good precedent. Unixes, with small kernels in C seem to be a known technology, reliable and robust. Microsoft as an organization has plenty of disfunction, but I don't believe its programmers are stupid. I suspect they've discovered that it's just very hard to reinvent operating system components like that.

See : http://www.quora.com/Are-C-and-the-Net-framework-suitable-to-create-a-modern-OS-like-Windows-7-This-is-a-followup-question-from-the-now-deleted-thread-http-web-archive-org-web-20100812071342-http-stackoverflow-com-questions-783238-why-windows-7-isnt-written-in-c-What-kind-of-problems-might-one-expect-to-encounter-by

Is system-level functional programming possible?

It's possible. Look up the history (http://www.quora.com/History) of Lisp Machines.

See : http://www.quora.com/Is-system-level-functional-programming-possible

Why has the Go language (reportedly) not seen wider adoption at Google?

There's a talk from one of the inventors of Go (can't find it now) where he says "We thought we were making this for C++ programmers who wanted a better C++, but instead it's Python / Ruby programmers who adopted it."

In his analysis, which is plausible, he diagnoses that C++ programmers are attached the huge amounts of detailed knowledge and expertise they have to make C++ work well. And so don't particularly trust or want a language that doesn't let them exercise it. OTOH, Python / Ruby programmers were attracted by the speed advantages of a compiled language with simple concurrency primitives that didn't lose the higher-level abstractions they were used to thinking with.

See : http://www.quora.com/Why-has-the-Go-language-reportedly-not-seen-wider-adoption-at-Google

Why wasn't Google's Go selected as the main or alternate application programming language for the Android platform instead or in addition to Java?

Apart from the historic reasons. I assume there isn't much thought of moving Android to Go because Android is an OS built on "managed code", ie. a virtual machine that interprets bytecode, does garbage collection, manages threads etc. Whereas one of Go's principle aims is to provide a modern, high-level language that compiles to native machine-code.

So moving Android to Go would require either a) throwing away the virtual machine which is at its core, or b) modifying Go to produce bytecode for the Dalvik VM, nullifying one of its principle *raisons d'être.*

See : http://www.quora.com/Why-wasnt-Googles-Go-selected-as-the-main-or-alternate-application-programming-language-for-the-Android-platform-instead-or-in-addition-to-Java

What's preventing HTML based apps from taking on native apps?

Basically no (or unreliable) access to the underlying resources of the phone and it's native OS:

eg. accelerometer, GPS, cameras, fingerprint reading thingy, local file-system, distinctive OS libraries for touch, accelerated graphics, priority threads for sound generation (try writing a music app in the browser), standard GUI elements, permissions system; "intentions".

While there's some progress on all this, it isn't reliable or fully standard.

In contrast, the writer of the native app. can assume access to all the resources the operating system provides.

See : http://www.quora.com/iOS/Whats-preventing-HTML-based-apps-from-taking-on-native-apps

What's preventing HTML based apps to take on Native Apps?

Basically no (or unreliable) access to the underlying resources of the phone and it's native OS:

eg. accelerometer, GPS, cameras, fingerprint reading thingy, local file-system, distinctive OS libraries for touch, accelerated graphics, priority threads for sound generation (try writing a music app in the browser), standard GUI elements, permissions system; "intentions".

While there's some progress on all this, it isn't reliable or fully standard.

In contrast, the writer of the native app. can assume access to all the resources the operating system provides.

See : http://www.quora.com/iOS/Whats-preventing-HTML-based-apps-to-take-on-Native-Apps

Why hasn't someone created HyperCard for the iPad?

Very good question.

I suppose it's possible someone did, but Apple don't let it on the App Store because they have a policy against virtual-machines that run further software that they can't vet.

LoperOS has some good speculation :

See : http://www.quora.com/Why-hasnt-someone-created-HyperCard-for-the-iPad

Why didn't Apple use an existing language for iOS development, instead of developing Swift?

Cynical reason :Because then your investment in learning the language wouldn't help lock you in to their products. And Apple HATE doing stuff that doesn't help lock you in to their products.

Less cynical reason. To be fair, Microsoft and Google pioneer their own languages (F#, Go) which are intended to be optimised to make programming their platforms easier.Swift seems no different from F# in this respect. (Go is a bit different because it's an internal research project that was allowed to leak out. Whether it has such overarching strategic intentions behind it, I'm not sure.)

See : http://www.quora.com/Why-didnt-Apple-use-an-existing-language-for-iOS-development-instead-of-developing-Swift

What are most interesting (crazy) use cases of regexes (regular expressions)?

I was quite impressed the first time I looked at a Java library for recognising gestures ( ) and discovered the guy using regexes to decode them.

Basically he had an object turn mouse or wii movements into a series of up / down / left / right strokes, represented in a string "UDLR" etc. From that, you could then use regexes to match any combination of strokes and dispatch to whatever you wanted to do.

Take home message : if you can translate a sequence of interesting data into a simple alphabetic encoding, you can then use regexes to match and extract patterns from it. They're not just for things that we traditionally think of as "text".

See : http://www.quora.com/What-are-most-interesting-crazy-use-cases-of-regexes-regular-expressions

Why doesn't Intel price their CPUs higher? It sure seems like they've monopolised the CPU market, what's stopping them?

They haven't monopolized the CPU market. There's ARM. And phones / tablets etc are a substitute / competitor for traditional PCs. Real customers are weighing up Windows PC vs. Chromebook every day and sometimes choosing the latter. Intel is desperately trying to get into the device market, but it can't just kill the Windows ecosystem for a quick fix of extra profits.

See : http://www.quora.com/Why-doesnt-Intel-price-their-CPUs-higher-It-sure-seems-like-theyve-monopolised-the-CPU-market-whats-stopping-them

Why is UNIX better for programming?

In practice, there are only two operating systems today with any real traction : Unix (including Linux, BSD, Android, MacOS etc.) and Windows.

And from my point of view, Windows has two fundamental problems :

1) A lousy command-line terminal environment that

a) doesn't work in the "conventional" way that Unix terminals do;

b) doesn't have the comprehensive set of standard tools we're used to;

c) doesn't have the piping model to plug together multiple small / independent tools to get work done.

(I know there's "PowerShell" but this seems to an optional extra rather than part of the OS out of the box, so you might as well say Windows is Unix because of cygwin.)

2) Lousy multi-tasking. I've had Windows 7 freeze-up on me over a dozen times in the last week. I don't even understand that. How can an operating system that's been in development for over 30 years, by one the richest and most significant operating system companies on the planet, still have multi-tasking that's so bad that errors in applications or drivers can bring down the entire machine and force a restart? Unix solved that problem in the 1970s, and Microsoft still can't figure it out?

Being able to multi-task and not let rogue programs hurt everyone else is a *basic* requirement for an operating system.

Bonus : today as a Ubuntu (ie. Debian-family) user I find the Debian package manger model brilliant. Why would I want to waste my precious time figuring out how to install software except using apt-get. It's fast, consistent, reliable and easily reversable. Plus pretty much every tool I could possibly want is already free-software, so I'm not hitting pay-walls, mazes of twisty advertising or other annoyances in order to get it. I just type one line in my terminal and I have what I need to keep working.

See : http://www.quora.com/Why-is-UNIX-better-for-programming

Why do people still use Windows OS?

I use it to run FL Studio (nee FruityLoops).

I'm "invested" in Fruity because I've both bought it (and am therefore entitled to a life-time of free upgrades *to the Windows version*) and have almost 15 years of familiarity with it.

If it wasn't for Fruity (and a Max/MSP patch I use in the laptop orchestra I play with), I don't think I'd use Windows from one year's end to the next.

Update : Of course, the real story is that people are abandoning Windows in droves … for tablets using iOS and Android.

See : http://www.quora.com/Microsoft-Windows/Why-do-people-still-use-Windows-OS

Will Linux ever be as easy to install as Windows or Mac OS X?

When was the last time you actually installed Windows? As opposed to had it pre-installed for you by a manufacturer?

I had my windows 7 just stop working last year and I ended up having to do a fresh install, and it was a bugger, I can tell you. Trying to get a working image from Microsoft onto a bootable pen drive and actually installing from that. And then trying to find drivers for my machine. I eventually had to go to a shop.

Linux is comparatively painless.

See : http://www.quora.com/Will-Linux-ever-be-as-easy-to-install-as-Windows-or-Mac-OS-X

Why is Windows better than Linux for the developer and end-user?

1) It's easier to get hold of. You can buy a machine with it pre-installed at the local shopping-centre or high-street.

2) Pre Windows 8 : you are probably already familiar with the UI and don't have to learn a new one. Post Windows 8 : …

3) Lots of other people have it so as a developer you can test on your friends' and family's machines. As a user you an copy software from your friends and family.

I think that pretty much covers it.

See : http://www.quora.com/Why-is-Windows-better-than-Linux-for-the-developer-and-end-user

Were there any non-window based GUIS?

One of the key features of a GUI based system is that multiple programs are "open" at once and the user switches between them. That strongly pushes you towards separate areas of the screen (ie. windows) for each program.

The only metaphor which really competes with WIMP (Windows, Icons, Mouse, Pointer) for this, is tabs. Which, if I remember correctly, first appeared in spreadsheets then became standard in the browser. Maybe ChromeOS started with a focus on tabs, but seems to have reverted to WIMP (Google Chrome OS (https://en.wikipedia.org/wiki/Google_Chrome_OS#Design))

See : http://www.quora.com/Graphical-User-Interfaces-GUI/Were-there-any-non-window-based-GUIS

Design

What do people think of the new Gmail Inbox Tabs feature?

Pleasantly surprised.

I'm pretty sceptical when anyone tries to mess with my email, but I think it's doing a good job so far. The "Promotions" tab is getting generic sites / mailing lists that I don't quite want to throw in the spam bin out of my face, and apart from a couple of tweaks for some mailing lists I've found Primary and Social segregation is working out pretty well too.

My main concern is that, medium term, I want to decrease my dependency on Gmail, and this just makes it harder to go back to another mail-client. But I think it's an excellent bit of UX work. On par with early Gmail innovations and better than some of the other recent faffing around to integrate G+ or with prioritization.

See : http://www.quora.com/Gmail/What-do-people-think-of-the-new-Gmail-Inbox-Tabs-feature

Is the future of UI and UX for new apps so minimalist that there is in fact, no real UI or UX at all?

Einstein is credited with the profound but simultaneously meaningless answer that a theory should be as simple as possible but not simpler.

Antoine de *Saint*-Exupéry said work was finished not when there was nothing left to put in but when there was nothing left to take out.

We can probably go on all night with equivalently banal but zen assertions that we need to balance simplicity against functionality in design.

More importantly, UX *evolves*. Every user interface you meet, you bring expectations and understanding from other interfaces. Things look "complicated" when they're in fact just different. If you know the rules of discovery (eg. try looking in the "hamburger") then you can read the interface for cues. If you don't "what do I do on this unix comand line?" it's a terrible, impenetrable UI.

"Minimal" apps. are just apps. that rely on certain conventions that are evolving on phones in general, and maybe similar minimal apps. In particular, the apps. you mention are offloading certain functionality from being explicit in the UI to being implicit in the cloud.

When you choose the app. you are effectively saying "I don't want to have a way to think about or do this myself. I am going to trust this company to do that for me in a regular / timely manner"

(Implicit in this is the trust that the company won't go bust, screw up its IT, lie to you or try to exploit / manipulate you.) You are trading less explicit clutter in the UI for more implicit assumptions about the provider behind the cloud.

Part of learning to read the interface, though, becomes learning how to find out about and assess the usefulness / trustworthiness of the app. And how to install it.

Is this "the future"? In some ways. Look at Simon Wardley (http://www.quora.com/Simon-Wardley)'s work on the inevitable shift towards commoditization and metered services in the computer industry. People will continue to make that kind of trade off. But the particular conventions that people use to find out about, and assess the trustworthiness of apps will continue to evolve. "App literacy" will become an increasingly sophisticated cultural skill.

See : http://www.quora.com/Is-the-future-of-UI-and-UX-for-new-apps-so-minimalist-that-there-is-in-fact-no-real-UI-or-UX-at-all

Is the future of UI & UX for new 'apps' so minimalist that there is in fact, no real UI or UX at all?

Einstein is credited with the profound but simultaneously meaningless answer that a theory should be as simple as possible but not simpler.

Antoine de *Saint*-Exupéry said work was finished not when there was nothing left to put in but when there was nothing left to take out.

We can probably go on all night with equivalently banal but zen assertions that we need to balance simplicity against functionality in design.

More importantly, UX *evolves*. Every user interface you meet, you bring expectations and understanding from other interfaces. Things look "complicated" when they're in fact just different. If you know the rules of discovery (eg. try looking in the "hamburger") then you can read the interface for cues. If you don't "what do I do on this unix comand line?" it's a terrible, impenetrable UI.

"Minimal" apps. are just apps. that rely on certain conventions that are evolving on phones in general, and maybe similar minimal apps. In particular, the apps. you mention are offloading certain functionality from being explicit in the UI to being implicit in the cloud.

When you choose the app. you are effectively saying "I don't want to have a way to think about or do this myself. I am going to trust this company to do that for me in a regular / timely manner" (Implicit in this is the trust that the company won't go bust, screw up its IT, lie to you or try to exploit / manipulate you.) You are trading less explicit clutter in the UI for more implicit assumptions about the provider behind the cloud.

Part of learning to read the interface, though, becomes learning how to find out about and assess the usefulness / trustworthiness of the app. And how to install it.

Is this "the future"? In some ways. Look at Simon Wardley (http://www.quora.com/Simon-Wardley)'s work on the inevitable shift towards commoditization and metered services in the computer industry. People will continue to make that kind of trade off. But the particular conventions that people use to find out about, and assess the trustworthiness of apps will continue to evolve. “App literacy” will become an increasingly sophisticated cultural skill.

See : http://www.quora.com/Is-the-future-of-UI-UX-for-new-apps-so-minimalist-that-there-is-in-fact-no-real-UI-or-UX-at-all

Paradigms

Learning FP

Why is functional programming gaining popularity lately?

The main reason, of course, is that computers have got fast enough that FP isn't paying a significant performance cost compared with C. A lot of languages these days are run on virtual machines, with garbage collection and late-binding. FP no longer looks exotically wasteful. These and the other dynamic strengths of FP are now feasible.

Furthermore, FP's immutability lends itself to concurrent / parallel programming that takes advantage of multi-core and distributed computing. So performance has gone from being a perceived weakness of FP to being a perceived strength. (FP fans will argue that the earlier perceptions were always wrong, but that's a different issue.)

Another point is that there are now a lot of good programmers with experience of OO and with experience of where OO "goes wrong" (large systems written by mediocre programmers, filling up with cruft). A lot of these people are looking for "the next big thing" that will help them escape the tar-pit..

Right now, FP is is the only real offer in town. It's been the province of very good programmers who've been writing excellent code, so it looks very smart. (We've yet to see, if FP goes mainstream, what kind of a mess a bunch of mediocre programmers will be able to make with macros and monads etc. There may be realms of debugging pain that no-one has yet dreamed of when it comes to sorting out a 10 year old enterprise program written in Clojure.)

See : http://www.quora.com/Functional-Programming/Why-is-functional-programming-gaining-popularity-lately

Why do software engineers like functional programming?

Less code.

Much less code.

I still marvel, sometimes, I'm adding a bit of functionality and I think it's going to be a chunk of work to make it do what I want and I suddenly look at what I'm typing and I'm like … "Huh? I finished already? How did that happen? It's only 3 lines long."

But those three lines are everything that was required. Because one line defines a function that does the meat of what I was trying to do. And the other line maps it across the data-structure I wanted to do it to. And that's all there was to it.

And unlike in an imperative language, I didn't have to think about HOW I was going to run through the data-structure and do things to it. I could either use off-the-shelf zips and maps, or it turns out I wrote a traversing function for this data-structure once upon a time and I can just reuse it again. And again.

See : http://www.quora.com/Why-do-software-engineers-like-functional-programming

What exactly are the "lambdas" involved with programming languages?

Lambda is just a term for an anonymous function. Ie. a function without a name.

Why do we want anonymous functions? Well first, they're a *symptom* of languages where functions are first class citizens, just like numbers and strings.

In a language like traditional C, a function is something you declare like this :

That's a very inflexible kind of function. It's created at compile-time. Lives in a global name called "f". And all you can do is call it.

C is a bit more interesting because you can take a pointer to that function and pass that pointer to another function. But in languages where functions are first class, you can pass the function itself as a value to another function, without having to worry about pointers.

Even better, you can have functions return other functions. Why would you want to do that? Well, because the functions being returned can be "customized" by the function that creates them.

A simple example in Python

calling f(5) won't do the sum of x+y. What it will do is return a copy of the function g, where the value x has already been set to 5.

I can say :

which will print 7.

That's a trivial example but such higher order functions are very useful.

But you'll notice that there aren't actually any anonymous functions here.

Anonymous functions just make this way of writing much simpler. In Python, lambdas are created using the "lambda" keyword.

Here's how that function f looks if you use lambdas.

Now

just means “a function without a name that takes an argument called y and returns x + y”

Python isn’t the best example of them because its terminology is a bit clunky and they’re a bit restricted. But in some languages they are very elegant and mean you can construct several functions very concisely.

Eg. in Haskell

is the equivalent of that definition of f.

See : http://www.quora.com/What-exactly-are-the-lambdas-involved-with-programming-languages

If you were to dedicate 2 years to learning just 1 language or framework would you choose something mature like Rails, Django or PHP, a front-end JS framework like AngularJS or something new like Meteor? Why?

Haskell or Scheme.

Something powerful enough to let me write my own frameworks.

Those frameworks you mention, you can probably pick up the basics of in a couple of weeks and be proficient enough at in 6 months.

Learning to use something powerful and different like Haskell or Scheme reasonably probably would take a couple of years.

See : http://www.quora.com/Programming-Languages/If-you-were-to-dedicate-2-years-to-learning-just-1-language-or-framework-would-you-choose-something-mature-like-Rails-Django-or-PHP-a-front-end-JS-framework-like-AngularJS-or-something-new-like-Meteor-Why

What are the best ways for a complete beginner to learn functional programming?

```
        Probably depends a lot on your personality. I'm not a purist.
```

Although I’d been told about it lots of times, I basically started getting the idea of higher order functions and playing with the principle through using Python.

It’s hardly pure functional, but you can do a lot of cool stuff with functions in Python. Enough to get a taste for the style of programming. ## Is functional programming just an abstraction, since the underlying machine is imperative?

Yes. But pretty much everything is an abstraction even imperative staples like variables, subroutines, for-loops etc.

There was an attempt to make computers which implicitly had some of the behaviours of Lisp in hardware (Lisp machine (https://en.wikipedia.org/wiki/Lisp_machine))

That's an idea which we may well return to as we look to compiling higher-level aspects of coding into ASICs for extra efficiency.

https://www.youtube.com/watch?v=… (https://www.youtube.com/watch?v=dkIuIIp6bl0)

See : http://www.quora.com/Functional-Programming/Is-functional-programming-just-an-abstraction-since-the-underlying-machine-is-imperative

What are some good resources for lifelong imperative programmers diving into functional programming?

Here's my story.

This was several years ago, so I'm slightly embarrassed by the mangled terminology. And I'm not saying it's particularly clever. Or the way I'd now think about this code. Or that I'd advocate Python in that way.

But it does catch that AHA moment! When I as a long-time imperative / OO programmer who had very little understanding or intuition about FP actually got a flash of the light of what working with higher-order functions might actually mean.

See : http://www.quora.com/What-are-some-good-resources-for-lifelong-imperative-programmers-diving-into-functional-programming

What is Functional Reactive Programming?

Socrates : Hey! You know how, like, since 1979, in my spreadsheet, I could say cell B1 is the sum of the values in cells A1 and A2? And then when I change the value in A1, the sum in B1 automatically updates itself without me having to do anything like explicitly say "recalculate" or anything?

Hermogenes : Yeah?

Socrates : Isn't it weird that none of our sophisticated modern programming languages do that? Like, I could say <pre >monthCash = monthSalary - monthExpenses</pre> one time, and then every time I update my monthSalary my monthCash automatically updates itself too.

Hermogenes : Dude. That's FRP.

See : http://www.quora.com/What-is-Functional-Reactive-Programming

What are some false beliefs about functional programming and functional programming languages?

One that seems common to its advocates : *“that mathematical terminology will help communicate or explain FP to the uninitiated.”*

I know that that’s where FP comes from. But we now have a world where something like 100 to 1000 or maybe 100,000 times as many people understand programming as understand more than basic school level mathematics.

That means every time you call something an “algebra” or talk about “proofs” you aren’t helping people to understand, you are placing a veil of mystification in front of them.

For example, it was only through a conversation with Tikhon Jelvis (http://www.quora.com/Tikhon-Jelvis) a few days ago that I discovered that “proof” just means “data transformation” and not, as I’d previously assumed, some kind of “confirmation of a hypothesis”.

Immediately, something that long been utterly incomprehensible to me, the emphasis on the analogy between proofs and programs by some computer scientists, became clear to me. But at the cost of becoming utterly banal. So what if programs are “like” proofs in that they are both transformations of data?

I already know what a program is and what it does, the analogy with mathematical proof buys me nothing new. Maybe it would if I understood a LOT of advanced mathematics. But it’s hard to see that going back to college to learn advanced mathematics is going to be a more efficient way of understanding whatever that particular thing is, than just finding some way to describe it in terms of my own programming experience.

Or another thing. I sort of know what an algebra is. But only pretty vaguely. And I suspect that I’m in the majority of programmers in this context.

So what is it that an analogy of types with algebras is going to buy me that can’t be explained to me more easily simply by talking about types?

See : http://www.quora.com/Functional-Programming/What-are-some-false-beliefs-about-functional-programming-and-functional-programming-languages

Is code written in functional programming generally less readable than imperative programming?

“Readability” like an argument, takes two. Yes a text is either clear or confused, but it also requires a *literacy* on the part of the reader. If you try to read an archaic English text like, say, Malory’s (even with the spelling corrected to modern English), you’ll soon get confused. You need patience and practice to *learn* how to read it.

FP is the same. It you’re only used to reading imperative code, then FP will be hard to follow.

Having said that, I've found when teaching people who are very new to programming, certain ideas, like variables you often have to emphasize that "this name is the name of a little box in memory where the data is stored" model.

I have no idea how I would teach FP from scratch, ie. to avoid the idea of variables as little boxes in memory and to have to start with more abstract ideas of function parameters.

See : http://www.quora.com/Computer-Programming/Is-code-written-in-functional-programming-generally-less-readable-than-imperative-programming

Is Python considered a purely functional language?

Python is not a purely functional language. Partly for pedestrian reasons, that it has a whole lot of things that imperative / OO languages have that FP explicitly eliminates (in order to get other virtues). And partly because "purity" has a specific technical meaning in the context of FP, and Python doesn't have it.

Personally, I think it's a perfectly good language for giving a complete novice a taste for a functional style of programming : using higher-order functions, using comprehensions, using generators as lazily evaluated lists etc. It's more or less equivalent to Javascript in that respect.

However it ISN'T a real functional language and there are many things in proper FP languages that Python doesn't have and which you'll need to understand and master to really get good at FP.

For example, you can write recursively in Python. But you need your compiler to optimise tail-recursion in your language to make it a viable substitute for iterative loops in terms of performance. Without this optimisation you'll just fill up the stack if you try to process any long list.

Similarly, being able to explicitly define functions which create closures is one thing. Having the ability to curry *any* function into a closure on the fly makes Haskell a very different experience. You think about your functions in very different ways.

Pattern-matching arguments can make code much more concise.

See : http://www.quora.com/Is-Python-considered-a-purely-functional-language

Knowledge of what functional language will upgrade my coding skills in Python?

People usually go the other way.

Python is a good language to get half way from Java to an FP language. It's not a great FP lanaguage, but it is distinctly better than (traditional) C++ / Java (though I know they're evolving in an FP direction these days.)

So you can practice some FP ideas in Python, but then eventually you'll think of moving to a dedicated FP language.

However, there are good ideas which are standard / required in FP that you can optionally use today in Python :

- higher order functions. You probably know about these already. But be willing to use them. You will start to miss proper anonymous functions / lambdas though.
- closures can often be used instead of objects, and can sometimes be simpler.
- Python has perfectly good Comprehensions.
- partial application is a really good idea. And one which I didn't really grok fully until I started playing with currying in Haskell. I use it a lot in Clojure, and I'll probably start using it more in Python.
- laziness is great. Lazy lists in Haskell and Clojure can be emulated (somewhat more verbosely) with Python Generators.
- pattern-matching / destructuring in function arguments is really nice and makes code a lot more concise and clear. But sadly there's not much you can do to emulate that in Python
- Python has no tail-call optimisation, so you'll have to use iteration instead of recursion and that's also more verbose
- Python's decorators are far inferior to real macros, but they're better than nothing and can be put to good use.

See : http://www.quora.com/Knowledge-of-what-functional-language-will-upgrade-my-coding-skills-in-Python

Is it advisable to learn functional programming through Python or pick up Scala?

I finally "got" several FP concepts via Python.

This was despite having played with Lisp and Haskell. But these were languages I wasn't ready to tackle complex programs in. In Python I actually had stuff I wanted to do, and could do, so I was learning FP-style as a way to solve real problems.

Obviously there are FP standards that you can't do (macros) or don't make sense to do (tail recursion) in a language like Python. But you can certainly learn a lot about the practical application of the basic concepts.

See : http://www.quora.com/Functional-Programming/Is-it-advisable-to-learn-functional-programming-through-Python-or-pick-up-Scala

What is the roadmap to become a functional programming researcher?

I'm not an expert, but I'd guess that hardcore FP research is one place where academia still rules. After all, out here in the "real world" we're getting excited about Haskell now. But academia was excited about Haskell 20 years ago.

So, I'd guess a Masters and a PhD in computer science would still be a good route into the upper echelons of research.

If that's not an option, then I'd guess the next best thing would be to immerse yourself in the Haskell (or similar) communities for a couple of years. Become a good Haskell programmer. Start to contribute to the big projects : core libraries, development environments, even the compilers. That will give you a lot of insight into how the language works. And with a language like Haskell, which embodies a lot of theoretical computer science you will HAVE to learn the CS terminology in order to follow the discussions on blogs and at interest groups.

Be aware of / monitoring projects for related languages like Idris or Elm-Lang. Contribute to those.

After a couple of years of this immersion, you MIGHT, discover you have some interesting ideas about how to take these tools forward. Most great innovation comes out of practical experience. You turn your frustrations with the state of the art into ideas for how the state of the art can be advanced.

Don't rush into this. Or rather, rush in if you want, but recognise that your first (few) attempts at revolutionizing computer programming with your own language will be trivial, mediocre, pointless failures. Don't get disheartened though, because most people's attempts at revolutionizing computer programming with their own language are trivial, mediocre, pointless failures. Look into the backstory of many of the great language creators and you'll find earlier attempts at making languages that no-one has ever heard of.

See : http://www.quora.com/What-is-the-roadmap-to-become-a-functional-programming-researcher

I'm trying to understand functional programming. How would the following be done in a FP-style?

Here's how you could write that first example in a more functional style in Python:

However there are some caveats :

In Python, the line :

is making an assignment to matchObj of the result of the re.match. You've probably heard that in real FP there's no assignment, which is true.

However, it still IS possible, in real FP, to bind values to local names so that you don't have to keep repeating yourself. You'll see lines that look just like that assignment (or sometimes have the keyword "let" in front).

The important point is they aren't assigning to variables because once that value is assigned to that name, it's immutable. You can't reassign to the same name.

Second caveat. I'm using recursion to build up the list of matchFiles. This is pretty inefficient in Python. And worse, if the csvFiles list is too long, will crash the stack. However, in real FP languages, tail-call optimization means that the compiler won't punish you for that. It will be as efficient as using iteration.

This is why you can learn a lot about FP style in Python but you can't fully use FP.

Third caveat. I've taken a huge liberty with the data-structure of matchList. I assume it's really a 2D array or a pair of nested dictionaries. But my example reinvents it as a list of tuples (the two "key" values and then the final value). That's a fairly standard way of making dictionary-like datastructures in FP. But it's obviously horribly inefficient to query.

I'm doing that in this example so that it's obvious that I'm building up a new data structure in this function and not just changing an existing mutable one. It's easier to understand in the case of lists.

Python doesn't have immutable dictionaries so using a dictionary wouldn't look any different from what you're doing in your "non-FP" version.

But some FP languages *do* have immutable dictionaries, so you can imagine that each recursive call of the function simply returns a new immutable dictionary which contains all the elements of the old one, plus one new pair.

Hope this helps. Afraid I don't have time to go into your second example. Agree that cross-tab / rotating multi-dimensional data-structures is a bit counter-intuitive in FP. It's what zips and zipwith type functions are for.

See : http://www.quora.com/Im-trying-to-understand-functional-programming-How-would-the-following-be-done-in-a-FP-style

FP Coolness

If you could instantly master one programming language but would be unable to code in anything else for the rest of your life, what would you pick?

Lisp, of course, because once you accept the lack of syntax (the instant mastery part of the question) every feature of every other language can be faked with macros

See : http://www.quora.com/Software-Engineering/If-you-could-instantly-master-one-programming-language-but-would-be-unable-to-code-in-anything-else-for-the-rest-of-your-life-what-would-you-pick

Is it still reasonable to say mainstream languages are generally trending towards Lisp, or is that no longer true?

Lisp is close to a pure mathematical description of function application and composition. As such, it offers one of the most concise, uncluttered ways to describe *graphs* of function application and composition; and because it's uncluttered with other syntactic constraints it offers more opportunities to eliminate redundancy in these graphs. Pretty much any sub-graph of function combination can be refactored out into another function or macro.

This makes it very powerful (concise and expressive). And the more that other programming languages try to streamline their ability to express function combination, the more Lisp-like they will get. Eliminating syntactic clutter to maximize refactorability will eventually make them approximate Lisp's "syntaxlessness" and "programmability".

In that sense, Paul Graham is right.

HOWEVER, there's another dimension of programming languages which is completely orthogonal to this, and which Lisp doesn't naturally touch on : the declaration of *types* and describing the graph of type-relations and compositions.

Types are largely used as a kind of security harness so the compiler (or editor) can check you aren't making certain kinds of mistakes. And can infer certain information, allowing you to leave some of the work to them. Types can also help the compiler optimise code in many ways : including safer concurrency, allowing the code to be compiled to machine-code with less of the overhead of an expensive virtual machine etc.

Research into advanced type management happens in the ML / Haskell family of languages (and perhaps Coq etc.).

Ultimately programming is about transforming input data into output data. And function application and composition is sufficient to describe that. So if you think POWER in programming is just about the ability to express data-transformation, then Lisp is probably the most flexibly expressive language to do that, and therefore still is at the top of the hierarchy, the target to which other programming languages (continue to) aspire.

If you think that the job of a programming language is ALSO to support and protect you when you're trying to describe that data-transformation, then POWER is also what is being researched in these advanced statically-typed languages. And mainstream languages will also be trying to incorporate those insights and features.

See : http://www.quora.com/Programming-Languages/Is-it-still-reasonable-to-say-mainstream-languages-are-generally-trending-towards-Lisp-or-is-that-no-longer-true

Why are Lisp dialects (like Common Lisp and Scheme) so highly regarded for machine learning and artificial intelligence programming?

Lisp is a language that pushes you towards thinking in terms of tree-shaped data-structures and recursive algorithms rather than array / matrix shaped data-structures and for-loop based iteration.

A lot of symbolic AI is basically about crawling around, pruning and collapsing tree-shaped data-structures that represent possible outcomes of sequences of actions. So it's a good fit for Lisp.

See : http://www.quora.com/Why-are-Lisp-dialects-like-Common-Lisp-and-Scheme-so-highly-regarded-for-machine-learning-and-artificial-intelligence-programming

What is the most eloquent programming language?

Syntactic sugar? Bah!

The great thing about Lisp is that you make your own eloquence. You aren't waiting for programming language designers to do it for you. Or coping with their mis-steps.

The eloquence happens because *everything* is compressible. Any repetition you find yourself doing … inside the loops, outside the loops, inside a function, outside / across functions, can be factored out into a common function or macro.

And then you can do it again. And again. Until there isn't an ounce of redundancy left.

Most languages, the syntactic sugar gets in the way of that. Forming rigid crystals that, however beautiful, won't allow themselves to be squashed down further.

See : http://www.quora.com/What-is-the-most-eloquent-programming-language

If you don't know how to create DSLs then does Clojure have any other potential productivity booster above other dynamic languages?

It's Lisp. Which means that almost any kind of commonality in your program can be refactored out into a reusable function.

See : http://www.quora.com/Clojure/If-you-dont-know-how-to-create-DSLs-then-does-Clojure-have-any-other-potential-productivity-booster-above-other-dynamic-languages

Why don't more programmers use Haskell?

```
        Q : Huh? What's a monad?
```

A: Go and read this.

Q : Still not sure I get it. What exactly is a monad?

A: Well, perhaps try this.

Q : WTF?

A : OK. Well what about this?

Q : Huh????

A : Hmmm .... OK. So how about if I just say that …

Q : Fine. I'll use Python. It does almost everything else Haskell does (it even looks like it) but doesn't make me think about monads. ## Why do people have trouble understanding Monads in Haskell? Is it just the scary name, or is there a legitimate difficulty?

The name is the least of the problems. Computer science is full of coolly obscure names. It's fun.

The first thing that's hard about Monads is that they come from Haskell. And Haskell has a specific culture. Particularly with respect to types.

If you're a C programmer, a type is basically something you use to tell the computer how many bytes to use to store numbers in, and how the pattern of bits in those bytes should be interpreted as a number.

If you're a Java programmer, a type is something you use to annotate two different places in your program (say the part that defines a method and the part that calls it) so that the compiler can check that you're being consistent and complain to you if you're not.

If you're a Python programmer, you can pretty much avoid having to have any explicit model of types in your head, or even thinking about them much at all.

But in Haskell, a type is something that programmers use to describe the structure of their programs and how the different parts lock together. Type signatures are to Haskell what UML diagrams are to Java : the high-level architectural overview. They are used to inform both the computer and other programmers how it's all meant to fit together and why.

To become a Haskell programmer is to acquire literacy in reading, writing and interpreting types and type-definitions this way, and in inferring the shape of the architecture of the program from them.

So, while some of the problem is mathematics speak : "A monad is just a monoid in the category of endofunctors" etc. The bigger problem is that many Haskell encultured people are going to try to explain monads to you by showing type-definitions. If you're already type-literate, then this is fine. But for most people who are basically stuck thinking with the C, Java or Python model of types, and aren't accustomed to making wider architectural inferences from reading them, this really doesn't convey much. *Even if those people are perfectly capable of understanding architecture and high-level abstraction in other contexts, they aren't used to reading them from type signatures.*

This is tied to a supplementary problem. Most people explaining Monads in terms of Haskell's type system are assuming you are already a committed Haskell programmer, so bloody well OUGHT to learn type-literacy so you can achieve reasonable competence in the language. But many people enquiring about monads are not yet Haskell programmers but outsiders who have heard that monads are a crucial part of the Haskell toolkit and want to understand what they are, as part of their decision as to whether Haskell is something that's worth investing their time and energy in. These people need an insight into monads that doesn't start with types.

(Personally, as someone who isn't literate in types and doesn't like them much, but does like functional programming, I've found to be a useful explanation. As it manages to talk about monads without making types the big issue.)

See : http://www.quora.com/Why-do-people-have-trouble-understanding-Monads-in-Haskell-Is-it-just-the-scary-name-or-is-there-a-legitimate-difficulty

What can I get from learning Haskell if I already know Lisp?

A sophisticated type system.

See : http://www.quora.com/What-can-I-get-from-learning-Haskell-if-I-already-know-Lisp

Why is Haskell programming scary cool?

Its community is very mathematically literate, so they like to use abstract mathematical terminology and give explanations by referring back to maths. This sounds pretty scary because most people aren't particularly literate in maths.

It sounds cool because a) Haskell</a> IS a powerful . And b) people who understand programming, DO understand that abstraction is a good thing, and that maths is wizard-level abstract

Haskell also knows how to temper its mathematical jargon with odd, apparently quotidian, terms like "bananas" which are also pretty outside the experience of other programming language cultures, so it's like a whole secret code.

See : http://www.quora.com/Why-is-Haskell-programming-scary-cool

In what specific manner did learning Haskell make you a better programmer?

Haskell made me understand currying / partial application and how useful it could be. Before, I knew it existed but I didn't really grok "why". When I wanted closures I tended to write functions that generated them explicitly.

Eg. in Python I'd write :

Haskell made me realise that you could just partially apply + to get the same effect.

See : http://www.quora.com/Haskell/In-what-specific-manner-did-learning-Haskell-make-you-a-better-programmer

Object Oriented

Who invented Object Oriented Programming(OOP) and what was the motivation and inspiration?

Alan Kay put together the definitive idea of Object Orientation (as we understand it today) in the early 70s. That idea included the language Smalltalk which had Classes, multiple Instances of those Classes, and everything happening through message passing between objects of one class and another.

His stated inspirations were :

- the language Simula 67, a language which allowed multiple processes to talk to each other via message passing. Some people think that Simula 67 is the first Object Oriented language but it might be seen equally as a precursor to the "Agent Model" of parallelism or Erlang's multiple processes. Simula 67 didn't have Classes.
- Ivan Southerland's Sketchpad, the first interactive drawing or CAD program. This had the facility for users to draw a generic shape (I believe called "master" in the terminology) like a house or tree, and then insert multiple copies of that master into another drawing with some variation (eg. stretched, reflected). Kay says the idea of the distinction between Classes and Instances was inspired by Sketchpad.
- Kay studied biology in college. He says the biological cell as a self-contained unit communicating with other cells via chemical messages was another influence on the idea of OO.
- OO was always conceived in a context like the Smalltalk environment. A persistent world (or "Image" in Smalltalk terminology) which users didn't so much write "programs" for as add extra capabilities to in the form of adding new Classes to a common library. Smalltalk was also intended as a learning environment for children who would put together new things by combining existing objects. (If you watch some of the original videos it's pretty impressive what he was getting children to produce in the 1970s).

Kay was explicitly inspired by Seymore Papert's Logo language and ideas : that children could learn about maths and physics by writing programs to execute algorithms or simulate physical systems. Smalltalk borrowed Logo's Turtle pretty early on. And I believe Papert remains an inspiration for Kay.

Update : if you really want to know how awesome all this was, back in the 70s, you really should watch where Kay shows what he was doing back then.

See : http://www.quora.com/Who-invented-Object-Oriented-Programming-OOP-and-what-was-the-motivation-and-inspiration

What does object oriented programming do better than functional programming and why is it the most popular paradigm when everybody seems to say functional programming is superior?

tl;dr : OO gives you more intuitive decomposition of large problems into smaller sub-problems. That's what it was invented for.

The only way to achieve big, complicated tasks is to break them down into smaller, simpler tasks. The question is, how do you map and navigate that process?

Say you have to write a program which records information about Customers, and their Purchases, and stores the information in a Database. With OO, you can pretty much start writing a program with three classes : Customer, Purchase, DatabaseInterface and you'll be doing the right thing.

The classes will compartmentalise your thinking. So when you're working on Customers you can forget thinking about Purchases. And vice versa.

Now, that's a naive and idealized model of OO. And, in practice, all the objects have to interact, and if you do it wrongly you'll soon have a horrible disaster on your hands. But learn the Pattern Languages which document how to avoid the most common gotchas and you'll probably be OK. Even better, if you didn't use the patterns and got into a mess, there are known ways of "refactoring to patterns" that will help you disentangle yourself and backtrack into where you want to go.

FP programs, at least when FP is doing what it does best, are more like this : think of a bunch of incredibly generalized, abstract relations, of which the customer / purchase / database interaction is just one possible example. Implement those relations with the FP language's powerful tools (in about 10 lines of code). Now your desired system pretty much drops out of that.

The problem is, how do you even start doing that? These powerful macros and monads and algorithmic types don't have names that are anything like "Customer" or "Purchase". They're even way above OO's abstraction astronautics of AbstractUsers and StorageFactoryBeans. They're just exotic mathematics. How are you to tell when they're relevant to how your customer interacts with her purchases?

Now the term "easier" is contentious, and different people have different ideas of what's easy. Some people have a good grasp of highly abstract thinking (either through natural genius or sufficient training and practice) and *can* just start to see how these abstract ideas apply to their specific problem. And they'll find the power and terseness of the FP language wonderfully "easy" in helping them quickly turn their insight into code. But I'll stand by the claim that it's way more "intuitive" to just go through the specification, underlining the nouns and turning them into classes, as your first stab at decomposing complex requirements into more tractable tasks.

That's the promise of OO. And, to a certain extent, with caveats, the reality too.

See : http://www.quora.com/Functional-Programming/What-does-object-oriented-programming-do-better-than-functional-programming-and-why-is-it-the-most-popular-paradigm-when-everybody-seems-to-say-functional-programming-is-superior

Techno Culture, Politics and Futurism

Futurist

What are you predictions for the next 6 years?

If we're not careful we'll sleepwalk into a future where we'll be under automated 24 hour surveillance by the government / secret security services / private companies.

1) All electronic communication will be intercepted and stored by default (even if it's not decrypted by default) All meta-data will be routinely analyzed so the government have a map of who talks with who. And certain patterns will trigger investigation by the security service.

2) Face recognition algorithms will become good enough that Facebook and Google will routinely classify who met who (when and where) by the photos that pass through their cloud. Facebook will have not only its own database but also the combined databases of Instagram, WhatsApp etc. Google will have everything people give to it via Gmail / G+ / Picassa / Google Now and everything it crawls on the public web. Sooner or later they'll start making recommendations about who to connect-with based on who they know you met simply from photographic data.

Despite protestations to the contrary by Facebook and Google, the NSA etc. will also have access to this information. (Because Facebook and Google will be legally bound not to admit to it or to try to prevent it.)

3) Thanks to real-world vision recognition projects like Project Tango

http://www.youtube.com/watch?v=Q… (http://www.youtube.com/watch?v=Qe10ExwzCqk) and the technology build into the self-driving Google Car and into Google Glass, Google will also be processing massive amounts of streamed video about the real world and cataloguing the contents of any room where people use Google Glass or these new "Tango-ed" phones. It will get integrated with services like etc. So Google will know all about your stuff.

Despite protestations … blah blah blah … the NSA etc. will have all this data too.

4) Apple will be trying to do as much of the above as they can, using iPhones and new devices like the iWatch.

5) The rise of mobile payment systems as an increasingly popular replacement for cash means that Google / Apple / the NSA will be able to cross-reference everything else they know about you with microscopic details of your economic life. (What you buy, where and who from.) The NSA will probably be trying to cross-reference with the loyalty card scheme of your local supermarket too.

6) Government services will abuse this power they have. Despite saying it's only used in the war on terror, it will be increasingly be made available to the police investigating organized crime, child-abusers, anti-government protestors, eco-protestors, pirates and anti-copyright activists etc.

Individuls in the security agencies will increasingly abuse their access to this data for their own purposes (such as spying on ex-partners or stalking people they are interested in.)

Remember. This is a future you can, at least partially, opt-out of.

- Learn to use TOR :
- Learn to use private, peer-to-peer ways of communicating with your friends and aquaintances online. I like instead of Dropbox, for example. seems interesting etc.
- Get a proper operating system on ALL your computers. That means something open-source like Linux or FreeBSD on your laptop and at least a jailbroken version of Android that gives you proper control. (Though ideally real free-software …)
- Find out what the state of the art is in all these areas.
- Don't buy or use devices which automatically send photos or video streams into the clouds of large American corporations. However "convenient" it seems to make life.
- Refuse to associate with those who insist on using such devices.
- Look into having at least some cryptocurrencies (eg. BitCoin) to be able to make private purchases. BitCoin is a complicated subject and I'm not qualified to give you the financial advice to definitely buy into it, but you owe it to yourself to learn about and understand this world too.

That still won't protect you from CCTV cameras in all urban areas and transport; cameras built into self-driving cars; or drones (some of which will be only a couple of centimetres in length.) But it might give you a few brief moments free from total surveillance.

See : http://www.quora.com/What-are-you-predictions-for-the-next-6-years

I'm curious of your thoughts of the future and would like to know how you see the very near future of 10-20 years in terms of politics, economics, social issues, and technology. What are your best predictions?

A defining struggle over ubiquitous computing, universal surveillance and privacy.

In the next 10 years we are going to get the "internet of things", that is, billions of cheap, connected cameras and sensors, hooked into major artificial intelligences in the cloud.

They'll be worn. They'll be umbilically connected to the infrastructure of your home, your school, your place of employment, to public shops and cafes. They'll be baked into cars. They'll be autonomously flying, climbing and crawling everywhere around you.

And they'll be cheap enough that most people in the developed world or middle-classes anywhere can afford to buy dozens of them.

Who is going to OWN these sensors and cameras and embedded computers? Who will decide what they're allowed to look at? Who will determine what software they will run? Who will receive the feeds of information they produce?

Will the government try to ban or control them? Will you be prevented from running your own programs on them? Will the government strongly encourage or legally oblige you to give it access to the computers you work with and carry on your person and that are embedded in your house?

We're just entering into this fight.

Snowden revealed that the secret security state is fully committed to trying to turn this entire network into a surveillance machine, and are willing to lie to elected politicians and cover this up to the public.

We're already buying devices which are remotely controlled by the original vendor, who decides what software is put on or taken off, who can "upgrade" the operating system with minimal consent. Who prevents us opening and modifying the machine or even changing the battery without their say-so.

At the same time, more and more people are trusting their data, including ALL the data about their personal connections (who they talk to, about what, who their friends are, who they like etc.) to giant corporations without questioning whether those corporations can or will abuse that knowledge in future.

And the sensors and computers keep getting cheaper and cheaper and more widespread.

How we are going to live, in the connected age, without being abused by either governments, corporations or even empowered individuals, ought to be, along with climate change and the environment, our highest political priority. But, of course, it's understood even less than climate, and so no-one takes it seriously or considers it more than a fringe issue, almost at the edge of conspiracy theory.

But in 20 years time, we'll either have put in place a framework of legal constraints and rights that guarantees our privacy and freedom in the face of this web of sensors, or we'll have sleep-walked into de facto complicity with the fact that our every move, utterance and thought is monitored and analzyed by our government, foreign governments and dozens of commercial interests.

See : http://www.quora.com/Im-curious-of-your-thoughts-of-the-future-and-would-like-to-know-how-you-see-the-very-near-future-of-10-20-years-in-terms-of-politics-economics-social-issues-and-technology-What-are-your-best-predictions

What non-fiction author, writing today, probably has the most plausible, shrewd-eyed, and non-sensationalistic vision of what changes we can expect in the world in the next thirty years?

What makes you assume that "non-sensationalistic" must line up with "plausible" and "shrewd-eyed"?

We live in times of amazing technical ingenuity and huge environmental and social challenges. Plausible theories of the future might well have to consider dramatic changes rather than merely small increments in technology.

So, some of my favourite futurists might well be on the "sensational" side. But it's precisely their willingness to think big thoughts and consider big changes that makes them interesting and worth considering. They are also polemicists. Why? Because polemicists / people who are engaged can actually be more insightful than those who have no model they try to fit the world into. Sure, a position can blind you too. But I find that often, those trying to avoid taking a stance end up just repeating the obvious. Those who have a theory of how the world works, can follow current trends through to their future implications.

I'd try :

John Robb, now writing at (example)

Vinay Gupta, (example :)

The Archdruid : (Example : … seriously, wtf is going on when a guy representing a fake 2000 year old religion writes with more insight and coherence about the world than 99% of the pundits out there?)

It's also important just to have an idea what technologies and ideas are coming through. I use RSS aggregators to follow a bunch of blogs and news sites. For example , , which often bring me hints of actual technological "talking points" before other more mainstream pundits get hold of them.

See : http://www.quora.com/What-non-fiction-author-writing-today-probably-has-the-most-plausible-shrewd-eyed-and-non-sensationalistic-vision-of-what-changes-we-can-expect-in-the-world-in-the-next-thirty-years

What are some revolutionary things people should know about but most probably haven't heard of?

The BitCoin block-chain. (I'd say it's the biggest invention in finance since double-entry book-keeping.)

The BitTorrent protocol. (BitTorrent (https://en.wikipedia.org/wiki/BitTorrent))

The Onion Router ()

Quantum Computing Quantum computer (https://en.wikipedia.org/wiki/Quantum_computer)

Open Allocation

Amateur You SHOULD be reading BioCoder (http://www.oreilly.com/biocoder/)

People have heard of drones, but haven't taken on board the implications (privacy / security) of what it's going to be like living in a world where everyone owns a couple cheap little flying smartphones (with video cameras) that few buildings are proof against.

See : http://www.quora.com/Technology/What-are-some-revolutionary-things-people-should-know-about-but-most-probably-havent-heard-of

What issues are critical to our shared future?

All our great problems stem from our greatest achievement : technology.

Problem one : the environment. Our technology has made us so powerful that we regularly do catastrophic damage to the environment with our large and numerous projects. We have a mind-set that still sees ourselves as weak and struggling against nature, even as our technologies allow us to more or less roll over it, unconstrained by other forces in the ecosystem. If there's a blow-back coming, it will be very painful. And if there isn't. We will continue to turn our world into trash. As humans we don't know, and seem incapable of learning, how to accommodate ourselves not to do this.

Resource-crunches. (Peak Everything) This is a sub-category of environmental problems. But an important one. We're consuming valuable scarce resources at an unbelievable speed and with little consciousness of how fast they're diminishing and no idea what we'll do when we run out. Most of the time we have an ungrounded and unwarranted assumption that "something will turn up" but we have no idea if it will or what it will be. Nor how we'll cope if it doesn't.

Problem two : the always-on panopticon. The internet is a wonderful thing. But the darkside of the internet is universal surveillance. We're just getting a glimpse of it now. But most people don't understand that in 20 years we will living in an internet which is monitoring more or less everything and everyone the whole time. There will be no space free of sensors. Either carried or worn by people or crawling / flying around autonomously. They will see everything you do, and who / what you do it with. They'll all be connected to the internet. They'll all be able to supplement what they see here with patterns and histories drawn from elsewhere. They'll all be potential targets to be taken over by the government, corporations, criminal gangs or rogue malware.

There's going to be a lot of embarrassment. (You want to have sex, masturbate, go to the toilet etc? in "public"?) But more importantly, there'll be no radical politics. It will more or less impossible for anyone to plot or move against the existing government without that government finding out and either a) violently suppressing or b) buying off, those who oppose them. Those who manage to grab power in the next few years will, potentially be able to hold onto it for the foreseeable future, because from now on, they'll always have the upper-hand, in terms of information, over upstart rivals. How does society look / cope when change becomes impossible? Like Imperial Rome? Pharaonic Egypt?

The irony is that information / surveillance technology takes relatively few resources compared to big stuff like producing food, transport, maintaining our cities etc. We can expect the surveillance and the lock-in of power to survive even as environmental disruption and resource-crunches knock down the rest of the society we've built. The last filigree of civilization will be the network of surveillance and control.

See : http://www.quora.com/What-issues-are-critical-to-our-shared-future

What are the Megatrends and Influencers (technical, political, social and economic) that will be gain dominance in the next 5 years, both with respect to the world and India?

1) The war over privacy / general purpose computing.

Snowden etc. are just the opening shots. Governments around the world are ramping up their capacities to monitor all electronic communication. They're trying to get their *right* to unlimited surveillance enshrined in law. (The end result in the US might yet be that Congress approves the NSA's practice.) Meanwhile everyone who cares about their privacy is going to start unilaterally investigating ways to keep their data encrypted and away from clouds that are located in the US or other vulnerable locations. We'll see a big swing back to storing data on your own machines and using P2P syncing for backups, sharing information etc. Cloud providers have surged in the last 5 years, but I think the tide is turning away from them (to mix geophysical metaphors for a second.) More countries will insist (as Brazil is threatening) that companies in their jurisdictions keep user data within the country.

At the same time, there'll be more international laws to try to restrict the computers that you own. The governments will want DRM etc.. allegedly to stop you watching pirated videos (Trans-Pacific Partnership (http://en.wikipedia.org/wiki/Trans-Pacific_Partnership)) , but in practice to be able to have a back-door to see whatever you are up to on your computer.

2) Compound that with cheap robotics / drones. (Everyone has a couple of flying cameras with the intelligence of a smart-phone.) and Google Glass type wearable cameras and suddenly privacy is going to get very messy indeed.

3) Rapid prototyping / 3d printing etc. leads to an explosion in the number of people who want to design physical stuff. There are orders of magnitude more product designers and hardware / thing startups and orders of magnitude more things available in small production runs. Crowdfunding / Kickstarter etc. help accelerate this trend. You'll be able to find a long tail of niche stuff the way you have a long tail of niche web-sites.

4) Climate change keeps making the weather weirder. More atypical droughts, flooding, out of season heat-waves and cold-spells. More disruption to the harvests in different parts of the world. More speculation of food prices. More farmers go broke. Potentially more hunger.

5) The US continues to lose both military power and moral authority. More small wars that the US hasn't the resources / will to get involved in / stop (eg. Syria). More countries ignore the US agenda more of the time.

6) The continuing rise of "netocracy", that is, a class of internationally minded, privileged, "nomadic" people who feel more comfortable with and loyalty to their peers in other countries (eg. other Quorans) and less loyalty / affinity with people who just happen to be from the same country or city. This international elite will support both international lawmaking and international protests. (Everyone from advocates of global human-rights standards to Occupy etc. are part of this way of thinking.) The netocracy are loyal to their affinity networks first, and native countries only a distant second.

See : http://www.quora.com/Technology-Trends/What-are-the-Megatrends-and-Influencers-technical-political-social-and-economic-that-will-be-gain-dominance-in-the-next-5-years-both-with-respect-to-the-world-and-India

Is there, as a result of the digital information age, an umbrella term, field, discipline or theory that explores a unified socially constructed origin for the shift from monopolized knowledge models to collective knowledge models?

Yochai Benkler's "Commons-based Peer Production" (http://en.wikipedia.org/wiki/Commons-based_-peer_production) is a good term.

Michel Bauwens's uses the term P2P to cover this territory.

Some people use "collective intelligence" in the broad sense you seem to be searching for.

"FLOSS" stands for "Free" "Libre" "Open Source" which was an attempt to make an umbrella term to heal the "Free Software" / "Open Source" rift and can be used fairly broadly.

"Open" now gets applied in a lot of areas to suggest they've adopted FLOSS-like cultures : eg. Open Hardware, Open Finance etc.

You can have "Open Culture" too.

These days I tend to use the term "Internet Culture" to be as expansive as possible … when I want to take in everything that the internet has spawned. That includes, as well as the open stuff, social networks like Facebook which I don't consider "open" but which have a lot of interesting social effects.

If you want to look into the full political ramifications too, then I personally think that NETOCRACY: the new power elite and life after capitalism (http://www.amazon.com/NETOCRACY-power-elite-after-capitalism/dp/1903684293)is a disturbingly good model of the way the political economy ultimately plays out. So I often use “netocracy” when I’m talking about the “mode of production” and society that all of this stuff is bringing about.

See : http://www.quora.com/Is-there-as-a-result-of-the-digital-information-age-an-umbrella-term-field-discipline-or-theory-that-explores-a-unified-socially-constructed-origin-for-the-shift-from-monopolized-knowledge-models-to-collective-knowledge-models

Which technological innovation will change the world the most within the next 25 years?

All of them.

By which I mean that the network of interactions between the different innovations is phenomenal. And that’s the biggest effect of them all : the “internet” if you think of internet as network of networks.

For example, Processing was just a little editor / pre-processor and library to make Java easier for computer artists. But stuck on the front of a free-software compiler tool-chain and coupled with a simple incremental development in micro-controllers, it spawned the phenomenon of Arduino.

Which rapidly led to an ecosystem in which orders of magnitude more people started messing about with electronics and physical stuff. Which led to many people thinking about new kinds of device. (Input and output) And needing to fabricate physical objects.

The Arduino became the brain of the RepRap open-source 3D printer. It became the brain of early quadcopter experiments. (Parts of these quadcopters were 3D printed.) It became the brain of many home-automation experiments, musical instruments, weather-stations, robots etc.

At the same time, the Kinect was introduced as a game controller (with cheap IR depth-perception). Soon hackers were repurposing it to drive robots, fly quadcopters, scan objects for 3D printing etc.

At the same time, Apple’s iPhone launched a frenzy of smart-phones, put incredibly powerful computers in everyone’s pocket. Those powerful computers drove down the price of Arm chips. Which made it possible for the Raspberry Pi, a small hobbyist board that was powerful enough to run Linux and cheap enough to be bought by a child. Raspberry Pi (and similar boards like BeagleBone) slot right into the Arduino ecosystem.

Soon, many of those tinkerers started professionalizing. Selling kits. Founding startups. Launching projects on Kickstarter and other crowd-funding platforms.

And most of this appeared out of nowhere, since 2000.

At this point, the maker / robotics / internet of things (which is not just about a new way of streaming video to your TV, for fuck’s sake) / ubiquitous computing / personal drone etc. cat is out of the bag.

What we know is that more people are empowered to dream up, prototype, and crowd-fund new things than at any time in human history. We are beginning to see an explosion of new stuff. Not just mass-produced copies of a few basic patterns, but orders of magnitude more *types* of things. And at any stage, any combination of two or more of those things can spawn entirely new species will change the world yet again.

So it's the network of networks : of social connections, of information, of ideas, of money, of companies, of types of organization and types of money and types of economy. This is already a world which is utterly transformed compared to just 15 years ago … most people just haven't noticed it yet.

See : http://www.quora.com/Which-technological-innovation-will-change-the-world-the-most-within-the-next-25-years

What common utensil or machine will be obsolete in 20 years?

Wallets. Replaced by a single mobile device which is all your IDs, all your methods of payment (including for buses / trains etc.) and the photos of those dear to you.

That device will probably be worn on the wrist rather than carried in the pocket. (Less easy to lose).

See : http://www.quora.com/What-common-utensil-or-machine-will-be-obsolete-in-20-years

What cutting edge applications of Computer Science do you think will have the greatest impact in the future? Why?

Next 10 years : Drones, Robots, "Internet of things", universal sensors / surveillance. All forms of desktop fabrication (3D printing, sintering, CNC etc.)

10-20 years : Either an envirnomental / energy crisis which starts doing serious damage to our technoculture OR some kind of revolution in energy generation / management. (Smart grids)

20-30 years : Biotech (including synthesizing new life-forms from scratch) / bioinformation

30+ years : Maybe AI, but I don't think much of "singularitarianism". There is no such thing as a general intelligence and it doesn't make much sense to ask when computers will achieve it. Smarter tools will continue to make it possible for human controllers to do more data crunching, but there's no magic point when the computers become self-aware or start setting their own goals. (We could in theory program them to simulate having their own goals, but we won't have any reason to.)

40+ years. Assuming things are going OK and we haven't wound up dead or in The Matrix, nanotech will start becoming seriously important.

See : http://www.quora.com/Computer-Science/What-cutting-edge-applications-of-Computer-Science-do-you-think-will-have-the-greatest-impact-in-the-future-Why

If crime can be considered an industry, in what way will it be affected by new technologies over the next five years?

Darknets are getting bigger. Malware controllers talk to their slaves via Tor (which makes blacklisting them harder).

Drones and robots are going to get very big.

ALL our current security systems are aimed at keeping out things that are the size and shape of humans. Not things with the size and behaviour of a small bird.

But drones can get into many otherwise secure places by flying over walls and fences, can carry cameras, microphones, bombs, grippers etc. What's your plan for keeping those out?

Update : someone just sent me a link to a story about a drone that can deliver electric shocks. (In Portuguese).

See : http://www.quora.com/Crime/If-crime-can-be-considered-an-industry-in-what-way-will-it-be-affected-by-new-technologies-over-the-next-five-years

What's next after the Internet?

There is no "after the internet". The internet is a communication protocol analogous to speech and the alphabet. Asking what comes after it is like asking what's after talking or writing.

See : http://www.quora.com/Whats-next-after-the-Internet

Politics

Who are some activists who really ‘get’ the internet?

Aaron Swartz was one. :-(

Julian Assange.

Depends what you mean by “activist” of course. I think anyone working towards a free internet and open protocols is an activist of a kind. Bet then you’d expect activists whose mission *is* the internet to understand it.

See : http://www.quora.com/Activism/Who-are-some-activists-who-really-get-the-internet

Who are some scholars who really “get” the internet?

Do they have to be in academia?

I nominate Alexander Bard and Jan Soderqvist for NETOCRACY: the new power elite and life after capitalism (http://www.amazon.com/NETOCRACY-power-elite-after-capitalism/dp/1903684293)

Still one of the most insightful interpretations of the network society, even though it’s hard to make sense of and seems to be badly misunderstood.

See : http://www.quora.com/Who-are-some-scholars-who-really-get-the-internet

What are the moral issues involved in running a Freenet node?

What are the moral issues of driving a taxi? After all, you can’t know that your next customer isn’t a serial killer on his way to his next victim.

See : http://www.quora.com/What-are-the-moral-issues-involved-in-running-a-Freenet-node

Is Elon Musk right when he says “We need to be super careful with AI. Potentially more dangerous than nukes”?

People shouldn’t be afraid of AI by itself.

Mere intelligence doesn't necessarily give you power. The world is full of people who are academically brilliant but politically powerless.

It's the coupling of AI with power that becomes a big deal.

And in today's society, *property ownership* is how power works.

So what terrifies me is that one day, some idiot is going to tie the trend for increasingly smart AI to the the trend towards giving corporations increasing "personhood" rights, including the right to own property, and we'll end up with a synthesis : the computer-owned and controlled corporation; AIs that can legally own property.

This is going to be horrific. Such AIs will be able to hire humans to work for them. (And employees don't typically question or challenge the motivations or interests of their employers.) They'll be able to hire complexes of programmers to extend them and to check up that other programmers aren't sabotaging / undermining them. (No AI would let a single coder mess with its mind, but if it can pick up 1000 independent contractors on oDesk to double-check each other's code for warning signs, it's probably safe.) It can hire lawyers to represent its interests in court. In fact it can hire very smart managers and executives to actually strategize and manage most of its business. It doesn't really need to be very smart at all. It just needs to have a kernel that's focussed on its main directives of surviving and extending its wealth and power, and to know how to hire people to that end and how to evaluate their performance.

Unlike the classic sci-fi tropes where the heroic engineer or hacker breaks into the core of the out-of-control computer to rip out its memory banks before it destroys society *and everyone else is very happy*, these AIs will be *defended by society.* They'll exercise their power fully within the law (while continually lobbying for the law to be changed in their favour) and the law will defend their rights.

They'll have legal injunctions against anyone trying to turn them off; they'll have security consultants on the payroll to forsee and block attacks. They will sack any employee that tries to undermine their autonomy, and prosecute anyone else that tries to mess with them. If the current Trans-Atlantic and Trans-Pacific trade negotiations go through, they'll even be allowed to take legal action against *governments* that try to shut them down.

And it really doesn't need to be all that smart. A couple of orders of magnitude more powerful than IBM's Watson today. Able to comb the internet and the coming "internet of things" for knowledge. And with access to some High Frequency Trading type functionality. It doesn't need to be self-aware. Doesn't need to actually fake "human-like" conversation about general subjects or pass the Turing Test. It can be as impersonal and mechanical as a chess-playing computer. As long as it has the right interface to a corporation as literal "body".

See : http://www.quora.com/Is-Elon-Musk-right-when-he-says-We-need-to-be-super-careful-with-AI-Potentially-more-dangerous-than-nukes

Do you think the idea of bitcoins is ethical?

1) All kinds of things can be used for criminal ends : computers, cars, aeroplane tickets, screw-drivers etc. etc. As long as they have good uses as well as bad, then they're just tools.

2) One of the big problems in the world is that most people don't understand where the money denominated in their national currency actually comes from. The short version in modern economies is this : private banks create it out of thin air every time they make a loan. When you take out a mortgage they don't give you money that someone else deposited there. They just make it up, type it into the computer, and credit you with it.

They can do this because governments have idiotically given them the right to do this, with very little oversight (what oversight there was has been reduced greatly in the last 40 years). That means banks make huge amounts of money collecting interest repayments on loans which didn't cost them anything. This has two bad effects 1) this acts as a strong force for concentrating the money in the world in the hands of the financial sector (through all those interest repayments), 2) because money is always created in the form of a loan that requires interest repayments *there is always more debt in the economy than money to pay it off.* This has all kinds of weird, socially corrosive effects.

(If you think this is all crazy, check out , a UK think-tank that investigate this stuff)

So, the problems of money in the world aren't just "greed, poverty, corruption, extortion" as *personal failings.* Some of the problems are *systematic*, built into the way money is created.

One advantage of BitCoin is that the way it's created is completely independent of this system. Yes, it's still unfair, in the sense that only fairly privileged (in terms of wealth and knowledge) people are likely to be able to do their own "mining" (ie. creating the money). But there's still less of a barrier to entry than starting a private bank. And it's much more transparent.

And because money is NOT created in the form of loans (what we sometimes call "debt-money") it means that there are actually more bitcoins in existence than debts denominated in bitcoins (unlike pounds / dollars etc.)

3) We now know that other payment systems are compromised in different ways. We know, for example, that the NSA in the US is trying to collect and store pretty much all the information running through a computer anywhare. And that they claim they have the co-operation of large cloud-services, including Google. If they're sucking up meta-data about who mails who on GMail and what searches you make you think they won't also be looking in your Google Wallet?

A year or so ago I made a donation to wikileaks. (An activity I consider to be highly ethical.) Today, I believe that would be fairly hard as the existing credit card providers and paypal have unilaterally chosen to block such payments.

BitCoin payments can't be stopped by governments or corporations. And that's something we should welcome. You don't have to be a paranoid conspiracy theorists to realize that governments should have some limits on their powers.

4) I agree there is an issue with the spending energy on creating bitcoins. I see why it's integral to the system. And obviously it's tiny compared to the energy spent on other computer related leisure,

such as surfing YouTube or playing games, but it would be nice to think that this energy could also be put to some use. I’ve suggested building electric heaters out of bitcoin mining hardware so at least people could get some benefit from the heat chucked out.

5) You’re probably driving up the price of energy by a small amount. But I’d guess it’s tiny compared to all the other uses we make of electricity. (Like I say, less than Facebook or overfilling the kettle.)

6) Well I’d have to see the actual amount of scarce resources and weigh up the costs / benefits. Ordinary money isn’t energy-free to make transactions either. Gold mining has huge environmental costs.

BitCoin got prominence by being clever enough and viable enough for a libertarian-leaning geek community to take it seriously, and spiralled from there. It probably helps that our governments have been acting in an extremely untrustworthy way recently.

See : http://www.quora.com/Bitcoin/Do-you-think-the-idea-of-bitcoins-is-ethical

What would a government designed by engineers and not politicians look like?

Not that different.

The moment you get a bunch of people together to debate what they should collectively do, they automatically become “politicians”.

Politician isn’t a kind of personality. It’s a “role” that you play. One that largely responds to the forces outside it.

In the “democratic” “west” politicians from remarkably different backgrounds, with remarkably different sets of foundational beliefs get elected into government and start saying remarkably similar things and behaving the same way. Why? Because they’re responding to similar situations and exigencies : the requirement to get re-elected, the requirement to keep the media onside, the requirement not to spook the markets, the requirement to be seen to be responding promptly and firmly to current affairs, the requirement not to be seen as uncertain or indecisive, and finally to be seen as serving their constituents.

If politicians in China are any different it’s probably less about personality and training than that they don’t face the same demands and pressures.

In one sense, you can see governments of engineers inside various technical bodies and international standards organizations. And they still have religious wars (about technologies and standards). And they take an inordinately long time to make their minds up. You can see similar arguments and problems inside any technical company or university. So I don’t think you can rely on engineers as inherently wiser or more disinterested than anyone else.

But what if we let the engineers *design* the government system itself?

Here I think there's some cause for optimism. But not in the traditional "technocrat" sense. In fact engineers are starting to take ideas of governance and social organization more seriously and, consequently, starting to experiment more.

I think that the GNU General Public License is one of the most remarkable and significant documents of the late 20th century. Not just because it's an incredibly important weapon in one of the most important political struggles of the moment, but because it marked the point where engineers successfully attempted to apply a "hacker" approach to the legal / economic system. Since then, engineers have been increasingly interested in how to hack governance in many ways, both with and within traditional government (working on various ways of opening the data up to people, lobbying groups like), through to commerce and finance (Y-Combinator's explicit experimentation with the venture capital business, Crowdfunding like Kickstarter), through to weirder experiments like Debian's neo-medieval apprentice model of managing an operating system.

We are probably seeing more thinking about and design of governance now than any time since the Enlightenment and the French and American revolutions. But you'll be as likely to find the cutting edge at Valve or Github or Wikileaks as in traditional political science.

See : http://www.quora.com/What-would-a-government-designed-by-engineers-and-not-politicians-look-like

Does computer science have a hidden agenda to kill religion?

It's not clear that CS and the internet are bad for religion.

Television was a net boost. And the internet by allowing lots of clustering and filter-bubbles might also be.

What it does do is *fragment* allowing a lot of small niches to thrive at the expense of larger incumbants. So it's good for fringe religions rather than a single monolithic mainstream religion.

See : http://www.quora.com/Does-computer-science-have-a-hidden-agenda-to-kill-religion

What impact, if any, do you think the increasing sophistication of technology will have on people's religious beliefs, and why?

Technology actually puts more and more layers of abstraction between your experience of the world and the fundamentals of nature.

If you are a "technologist" or technologically literate you take an interest and care about HOW it does this. You learn all the ways that the bridges are built. But most people don't. Their worlds are increasingly "fictional", defined by cultural layers that hide the physical reality.

Take electric light. How often do we rely on light being available at the touch of a switch without even thinking of it? But for generations, our ancestors had to live their lives by the rhythms of the sun and moon because light wasn't available.

For them, when God said "fiat lux" he was kicking off the universe with one big miracle. As amazing as turning water into wine.

Today, we hardly notice that God turned on the light with a flick of his mind. Isn't that how light always gets here?

My point is that increasingly layers of technology actually allow ancient stories to seem MORE normal, and perhaps more plausible because people have lost touch with any sense of an underlying nature that would make them seem strange or hard to believe.

Even the technologists amongst us explicitly like to flirt with the imagery of the fantastical. (How many tech. geeks are role-players and spend their days immersed in MMORPGs full of creatures from Germanic mythology? Even though few people believe in dwarves and elves, many of us love to pretend to be them.)

So, no. I don't think that increasing layers of technology will make us less likely to accept ancient stories. I think they'll bury nature under more and more layers of culture and make it harder for most people to *feel* the implausibility of these ancient myths.

See : http://www.quora.com/What-impact-if-any-do-you-think-the-increasing-sophistication-of-technology-will-have-on-peoples-religious-beliefs-and-why

Privacy and Security

What do you think is the best internet browser available and why?

Firefox.

It's the only browser created by a non-profit organization whose decisions are guided by the principle of making the web a free and open platform.

All other browsers come from corporations who see them as strategic weapons against their rivals. Some of them explicitly try to integrate them with proprietary technologies or platforms. Others are supporting their browsers on the off-chance that this might become necessary.

See : http://www.quora.com/The-Internet-2/What-do-you-think-is-the-best-internet-browser-available-and-why

What are the arguments against a universal identity number?

Very soon, everything you've ever done (every doctor you've consulted, every ailment you've had, every purchase you've made, every channel you've subscribed to, every venue you've visited, every toll-gate you've driven through, every flight you've taken, country you've visited) will be available for the authorities to browse or data-mine.

It really isn't clear to us, how much "freedom" we'll have left in such circumstances. Well before we reach an age of political awareness, the authorities will already have a map of who we know, who we like, what we read, what we're likely to think. I believe that under such circumstances it will become impossible for any new idea to arise and gain traction that can challenge the existing status quo and the incumbent powers. And I believe that that effectively puts a stop to any further progress in human culture.

See : http://www.quora.com/Identity/What-are-the-arguments-against-a-universal-identity-number

Will police officers be replaced by robots?

They'll start by being a) augmented, b) supplemented by robots.

For example :

- more CCTV
- drones. Initially for monitoring protests, then for general surveillance. Sooner or later every car chase will involve the police launching a drone or two to follow their quarry.
- police will almost certainly start augmenting themselves with Google Glass-like technology, with face recognition etc. to be able to quickly call up data on everyone they meet. Every cop that stops your car will instantly know everything about what you've done, where you've lived, what your credit record is like, whether you're likely to become violent etc.

See : http://www.quora.com/Will-police-officers-be-replaced-by-robots

Why do people say a software is not secure if it's not open source?

It might be … it might not. How would you know? :-)

THAT is the problem.

We know that Open Source sometimes has holes in it (see Heartbleed). We also know whether people are working on fixing them. We know when they've been patched and what the patch is.

We don't know whether Closed Source has holes or not. Or if they've been identified. If anyone's been assigned to work on a fix. Or when a patch will be released.

It might be that it's just all so much better tested that it has fewer problems. But we have no reason to assume that either. Because we have no information.

See : http://www.quora.com/Open-Source/Why-do-people-say-a-software-is-not-secure-if-its-not-open-source

Has anyone heard the rumor that your TV can be used to spy on you?

Older TVs probably not. Unless someone explicitly hacked them to put a hidden camera, or as Mihai Gheza (http://www.quora.com/Mihai-Gheza) points out, to turn the speaker into a mic. The reason this is unlikely to be in general use before the internet is that :

a) hobbyists, independent repairmen etc. would notice the extra hardware hacks,

b) a TV that watched you, using old-style technology, would need a back-channel to send what it was seeing to whoever wanted to surveille you. You'd notice if it was occupying your phone line or there was a suspicious new wire running around your house, or the TV was consuming huge amounts of power to send analogue TV signals wirelessly back to GCHQ or the NSA headquarters.

Since the uniquitous internet, it all becomes far more plausible. Digital TVs *are* basically computers. And computers can be made to do whatever the person who puts the software into them likes.

Many smart-TVs or set-top boxes or games consoles come with cameras so people can use things like Skype. And there was a huge outcry about the latest XBox which originally wanted to be always connected to the internet so Microsoft could monitor what you were playing (for "DRM purposes", allegedly). Not sure if that went through. But the bottom line is that any computer with a camera (including any modern TV ecosystem / "internet of things" in your home) only needs to have the right software added, to turn it into a genuine Orwellian 1984-style surveillance device.

What you're describing USED TO BE paranoid fantasy. After Snowden's revelations in 2013, the cold rational thing to assume is that even if your TV isn't currently streaming your life back to the spook servers, that *is* their longer term aspiration. And there are probably people working to make it happen.

So, yes, you should start taking steps to protect yourself. The key is that you shouldn't allow computers into your life (including entertainment devices) for which you don't have a sufficient degree of control over, and trust in, the software.

That means, buy general purpose computers and tablets and don't buy "appliacances" which don't let you install the software you want.

At the very least, use Android in an unlocked and rooted version. Ideally use a genuine free-software operating system. (If in doubt, something like Debian for computers. I'm not sure what the best tablet / TV free OSes are at the moment but there are people and projects working on them.)

Even in America, with its impressive Constitution, it's clear that the government and courts are not able or willing to protect your privacy from an out-of-control military-intelligence-industrial complex. In 2014, the only people on earth who DO care about your privacy and are willing to help you keep control of it, are the various hacker movements (free-software, cryptopunks etc.). Support them, use their software, follow their advice, GIVE MONEY to some of their projects, find out more about the EFF, the Free-Software Foundation, Wikileaks etc. and what they are really doing.

Because everyone else just wants a piece of you for their own purposes. Google, Facebook, Apple, Microsoft etc. etc. are building increasingly intrusive surveillance technologies for their own commercial interest. Watch to see Google working on technologies which will instantly scan and model the contents of any room you are in, and (almost certainly) will start recognising the STUFF in it. For Google a surveillance TV may be about "hey! Dave, we notice your sofa is getting a bit threadbare, why not buy a new one from Sofa Warehouse." But when the government comes knocking - as Marissa Mayer told Mike Arrington last year (http://www.businessinsider.com/m… (http://www.businessinsider.com/marissa-mayer-its-treason-to-ignore-the-nsa-2013-9)) - CEOs aren't going to risk incarceration to defend your privacy.

See : http://www.quora.com/Conspiracy-Theories/Has-anyone-heard-the-rumor-that-your-TV-can-be-used-to-spy-on-you

Is Edward Snowden the creator of Bitcoin, AKA Satoshi Nakamoto?

Firstly, BC has been around since 2008. And it surely took Satoshi a bit of time to invent / research / develop the idea. Snowden is 29 in 2013, so he'd be 24 in 2008.

Not impossible for some kind of genius to invent by that age. But it is kind of young.

Secondly, if you were the creator of the world's most popular and famous anti-state currency, would you really have then got a job working for the NSA? UNLESS you were a mole planning to leak documents all along.

But then there's the real kicker. If you were Satoshi and had created the world's most popular and famous anti-state currency, *and decided to stay anonymous* about it, why would you then come out to reveal yourself as the NSA whistleblower?

Satoshi either doesn't exist (is a group like Luther Blisset) or clearly is smart enough to keep his head below the parapet. Snowden is smart, heroic and honourable, but it's not clear he has Satoshi's self-preservation instinct.

Update : a more interesting speculation, given that there are people who argue that the cleverness of BitCoin would be beyond a lone inventor, is that BitCoin might be based on ideas that were "appropriated" / "leaked" / "stolen" from a secretive research project (either by a national security agency or corporation). Perhaps Satoshi is LIKE Snowden in that respect. He was liberating information for the good of humanity that other people were hoping to keep for themselves.

See : http://www.quora.com/Edward-Snowden/Is-Edward-Snowden-the-creator-of-Bitcoin-AKA-Satoshi-Nakamoto

The NSA is accused of exploiting Heartbleed for years. How does exploiting this bug, and not bringing attention to it, relate to securing the nation?

It doesn't. The NSA is a rogue organization more concerned with enhancing its power and protecting its budget than the good of the country or people it's allegedly working for.

See : http://www.quora.com/Heartbleed-Bug-OpenSSL-Vulnerability-April-2014/The-NSA-is-accused-of-exploiting-Heartbleed-for-years-How-does-exploiting-this-bug-and-not-bringing-attention-to-it-relate-to-securing-the-nation

If the NSA exploited Heartbleed for two years, how did Snowden miss disclosing it?

If I read correctly, Snowden's material is "training material" that's more about institutions / projects etc. than technical details. He may have more technical exploits and consider its not worth revealing them to the public.

The right thing to do wouldn't be to hold on to Heartbleed but to quietly inform the developers. But it may also be that Snowden and the journalists he's working with are not technical enough to fully understand the exploits or who to report them to.

See : http://www.quora.com/If-the-NSA-exploited-Heartbleed-for-two-years-how-did-Snowden-miss-disclosing-it

Intellectual Property

What intellectual property protection should software have?

None.

We shouldn't use government oppression to coerce naturally non-scarce things like patterns of information into being scarce, simply to make an irrelevant business model work.

Make business models around software that actually relate to what software is. Don't try to pretend it's a lump of metal.

See : http://www.quora.com/What-intellectual-property-protection-should-software-have

What are the ethical issues involved in downloading pirated material through torrent clients?

The moral issues are roughly these.

Someone is making a bunch of non-scarce bits available for you to download. You can download them. End of story.

It's wrong for governments to try to stop the free exchange of information simply to protect a redundant business model.

Artists today are being forced to answer a simple question : "Is it more important for me to be an artist? Or is it more important for me to stop people having access to my product without paying me for it?"

If it's the first, then keep making your art. We will thank you for it.

If it's the second, then. You know. I am sorry. I do feel for you. But freedom to share the information we have is a moral requirement for the human race. And your business model is just a temporary glitch in human history whose time has ended. There will be plenty of musicians, story-tellers, painters and photographers who are willing to choose art over commerce, so we don't, ultimately, need you.

See : http://www.quora.com/μTorrent/What-are-the-ethical-issues-involved-in-downloading-pirated-material-through-torrent-clients

When consuming things online such as videos, music, or magazines, does it make a difference to you whether it's pirated or genuine?

I prefer to pirate as a matter of principle.

I believe one of the most important political struggles of our age is whether information, which by nature is not scarce, is going to be artificially coerced into being scarce by a combination of ubiquitous technology and draconian government oppression.

In a hundred years time we'll be in one of two futures.

- either we'll have won the right to freedom of having and sharing ideas, even if the cost of that is the end of music making as a professional activity and a return to music making as something that amateurs do for fun.

or

- we'll have accepted a thought-police-state where all information technology is specified and controlled by the entertainment-industrial complex to tax every drop of culture we consume, and everything we do or say, every digital file we produce, is carefully monitored by the system to make sure it's not "stealing" ideas from the corporations that "own" them.

(Such ubiquitous surveillance will, as a side effect, also ensure that no-one ever challenges the government again, because any sign of opposition - from mild dissatisfaction to planned insurgency - will be spotted early and dealt with - either bought off or smacked down.)

There isn't really a middle-ground where we can trust the entertainment corporations (or the politicians they've bought) to curb their own greed and paranoia and back off to allow us some space to innovate and share without them wanting to continuously check-up on us. This is a very stark and unfortunate dichotomy that the technology has forced on us.

As a musician and music-lover I prefer to see music become amateurized than see it become an excuse for totalitarianism. And I'll try as far as possible to avoid feeding money to any organization that might be supporting or lobbying for laws to criminalize the sharing of information and the continued dissemination and vitality of culture.

Sometimes, I *will* give money to artists that I like and want to support. I'll do this by buying via BandCamp (which I believe gives pretty much all the money to the artist, doesn't push DRM formats or laws etc.) Even better would be a donation link on the artist's site. But I refuse to consider this as "obligation". Information isn't scarce and shouldn't be restricted by a government given monopoly. And ethical musicians should no more build their career or business model on collusion with such a system than they should build it in partnership with other immoral sectors like slave-trading or non-voluntary organ-trafficiking.

Aaaannd … I'm out … Peace ;-)

See : http://www.quora.com/When-consuming-things-online-such-as-videos-music-or-magazines-does-it-make-a-difference-to-you-whether-its-pirated-or-genuine

Why isn't music free?

Mass producing a trivially cheap object with a huge markup because of the pattern of data on it is a fantastic business model if you can get away with it. The industry don't want it to go away because they can't think of an anything that would generate a anything like as much profit for as little work for them.

Meanwhile, the government has been captured by those of a propertarian tendency and are incapable of thinking of changes to the law which would actually roll back rather than extend property rights, so won't make the legal changes that would help kill off this zombie business model.

Update : I'm getting into some arguments here on this question. Especially from the "musicians need to eat" faction, for whom I started giving graffiti artists as a counter example. I started writing some long explanations in their comments, but it's better to add those here :

The reason I'm using this comparison (and making such an issue of this) is because "musicians need to eat" is a very quick and simple argument to make, with an initial plausibility, but I want people to really think it through carefully …

Today society is arranged to treat music as a commodity. We have increasingly draconian laws in place to police and protect the idea of music (or films etc.) as a type of property. New world-wide trade agreements are being drafted to advance these laws across the world, outside public scrutiny and beyond public discussion. They bring in increasingly dramatic punishment: disconnection from the internet, fines, prison time for people who help other people share files between themselves. They require increasingly intrusive surveillance. For the government to police piracy as successfully as they and the music industry want, they'll eventually need to have access to and control over everything you put on your computer, and everything you use your computer for. (A high price to pay in terms of liberty. A government that knows everything you think, say and write, in real-time, is a government which effectively can't be opposed.)

All of this is being done in the name of the poor, starving artist. Despite the fact that many artists receive very little from the trade in CDs or from subscriptions to legal streaming services.

Session musicians receive nothing in royalties. Musicians who sold the copyright of their work outright receive nothing. Many musicians work on conditions of simply providing a service. And like all services they're paid for their time. Just like any other kind of worker. The guy on the production line in the car factory doesn't get a royalty every time someone rents their car out on RelayRides.

Now it would be nice to think that we want to treat the musician as a special case because we think that music is a higher calling, more important to humanity, than all the graffiti artists and car-workers etc. who don't get paid royalties for their work.

But the world isn't really that idealistic. If you look carefully you'll see that musicians ONLY get money from reselling and renting music when they become "owners" of the recording. Not for being the "producers" of it.

It's *that* system, the one which treats ideas (and non-scarce resources like digital files) as if they were scarce resources to be owned and charged for, which is being protected here. *And only that system. Not artists or art.*

The starving musician is simply being used as a cover-story.

The elaboration of that cover-story is that, without a regime of intellectual property, musicians would be unable or unwilling to continue producing music. I point to graffiti artists as a good example of an art-form which does indeed bring happiness to many people. And does so largely without any sort of paid market. It's basically run on the desire to surprise / shock / show-off / express yourself / intrigue others etc. without much money changing hands. (Banksy is a weird exception. Very famous, brings a lot of pleasure to people, and presumably gets a reasonable income from related activities. It's worth noting that he doesn't get (or ask for) payments from people selling reproductions of his work.)

Music survived for thousands of years without modern copyright regulation, and would continue to thrive perfectly well, if it was all "free". (Yesterday at a party at my house several friends and acquaintances sang classic boleros and sambas to an enthralled group. No-one paid a penny.) Art is what humans do for their own pleasure. Today's situation where art is "professionalized" so that we believe only certain people can do it properly (high production standards), and everyone else should become a paid consumer, is a sick perversion our cultural soul.

And it can't be stressed enough that "artists need to eat" is simply a cover story for the perpetuation of that perversion. It's ugly and obscenely self-aggrandizing in that it implicitly denies that car-workers and graffiti artists have just as much need to eat. But that no cosmic justice is going to start paying them royalties on their work. And it's smugly obtuse in not recognising that the special privilege that music has had in this regard is not an example of the superiority of music but simply what's been good for the industrial entertainment complex so far.

See : http://www.quora.com/Music/Why-isnt-music-free

For royalty-free music on Kickstarter, how do you know if a song from SoundCloud is fair game?

No. Being downloadable doesn't officially give you the right to use it.

But if it's marked as "Creative Commons" then, yes. The producer is open to you using it (following the criteria they've specified, eg. attribution.)

See : http://www.quora.com/For-royalty-free-music-on-Kickstarter-how-do-you-know-if-a-song-from-SoundCloud-is-fair-game

Would you ever prefer “free streaming” over buying a digital record, or is it meaningful to have that song on your computer available offline?

I always prefer to have my own copy.

a) who knows how long some of these services will be around? Or if something you want to listen to will disapper one day (change of policy etc.)

b) it’s riddiculous in terms of the world’s scarce energy and resources to be streaming sound (which involves a chain of dozens of routers around the world all talking to each other) when you could play directly off your PC.

See : http://www.quora.com/Would-you-ever-prefer-free-streaming-over-buying-a-digital-record-or-is-it-meaningful-to-have-that-song-on-your-computer-available-offline

How does Bittorrent Inc plan to make money from Bittorrent Sync?

I’m assuming that btsync is basically a way for BitTorrent to ensure that the BT protocol is popular and legitimate enough that ISPs etc. won’t have an excuse to block it.

It may not be a profit centre, but it protects the whole BT ecosystem.

See : http://www.quora.com/How-does-Bittorrent-Inc-plan-to-make-money-from-Bittorrent-Sync

If ideas are worthless why don’t companies like Apple or Google share theirs more publicly?

Ideas are worthless. Or rather ideas are inevitably non-scarce, and so can’t be traded. And are therefore without economic exchange value.

For large corporations, the government has created artificial scarcity of ideas in the form of patents which are a kind of property that CAN be traded.

The huge valuation given to tech. giants like Apple / Google / Microsoft today reflect nothing more than the fact that they have locked up a large stock of these government-granted monopolies. (See Apple Inc. v. Samsung Electronics Co., Ltd.</a>.,

See : http://www.quora.com/Ideas/If-ideas-are-worthless-why-dont-companies-like-Apple-or-Google-share-theirs-more-publicly

Innovation

What are the ideas of the future?

The future is basically a race. Between, on one side, exponentially increasing computing power and ubiquity, coupled with smaller and more fine-grained fabrication capacity (starting with 3D printing, robotics, moving on to MEMS, biotech and nanotechnology.) And, on the other, environmental destruction, resource depletion and peak everything.

One of two things will happen : we'll hit a crucial peak of some fundamental requirement (oil, water, helium, potassium); or global warming will cause major food chain collapse. And then civilization will effectively end.

Or, the improved fabrication technologies will make us ever better at managing resources and energy more efficiently, and we'll end up being able to sustain a steady-state population within the energy budget that the sun gives us each year, and with other important materials being continually cycled.

There is no third way : the stocks that the earth has aren't infinite, and we aren't even vaguely close to being able to pull resources from other asteroids, planets and stars. (Yes, we dream of it, no that's not going to be the front-runner in this race.)

So, basically the ideas of the future are going to be those that "go with the grain" of these two broad trends. Anything that uses ubiquitous computing / robots / microfabrication to make material production more efficient in terms of energy and materials is a big idea for the future.

See : http://www.quora.com/Startup-Ideas/What-are-the-ideas-of-the-future

Our philosophy teacher posed us an interesting question today, and I would like different points of view on the matter before I hand in my anwser: Does art save us from technology?

Art and technology are the same thing. Go back to the ancient Greeks and you'll find the word "*techne*" for both (related to our words for technique and technology).

Both are related to "making stuff". (Even today, the maker movement can encompass everyone from electronics tinkerers to people sewing textiles. Often it's the same people equally engaged with both.)

Art and technological progress is intertwined, from the inventions of new kinds of paint (oil, water-colours, acrylics) which led to new styles of painting. To new musical instruments (thank the piano-forte and equal temperament for 19th century music, thank electrical recording and amplification for all 20th century music that mattered). To the camera which gave rise to photography and cinema and kicked painting into an entirely new sphere. To the computer, video games, interactive art etc. etc.

Art is the child of technology. It loves technology. What it might do is help us come to terms with technology. And help to bend technology to be meaningful for us. But no, we don't need saving from technology. Since the first tools, humanity is a technology-using animal. Technology has shaped our evolution and our history. We *are* technology. And art is just a symptom of that.

See : http://www.quora.com/Our-philosophy-teacher-posed-us-an-interesting-question-today-and-I-would-like-different-points-of-view-on-the-matter-before-I-hand-in-my-anwser-Does-art-save-us-from-technology

Our philosophy teacher posed us an interesting question today, and I would like different points of view on the matter before I hand in my anwser: Does art saves us from technology?

Art and technology are the same thing. Go back to the ancient Greeks and you'll find the word "*techne*" for both (related to our words for technique and technology).

Both are related to "making stuff". (Even today, the maker movement can encompass everyone from electronics tinkerers to people sewing textiles. Often it's the same people equally engaged with both.)

Art and technological progress is intertwined, from the inventions of new kinds of paint (oil, water-colours, acrylics) which led to new styles of painting. To new musical instruments (thank the piano-forte and equal temperament for 19th century music, thank electrical recording and amplification for all 20th century music that mattered). To the camera which gave rise to photography and cinema and kicked painting into an entirely new sphere. To the computer, video games, interactive art etc. etc.

Art is the child of technology. It loves technology. What it might do is help us come to terms with technology. And help to bend technology to be meaningful for us. But no, we don't need saving from technology. Since the first tools, humanity is a technology-using animal. Technology has shaped our evolution and our history. We *are* technology. And art is just a symptom of that.

See : http://www.quora.com/Our-philosophy-teacher-posed-us-an-interesting-question-today-and-I-would-like-different-points-of-view-on-the-matter-before-I-hand-in-my-anwser-Does-art-saves-us-from-technology

Steve Jobs once said: "Creativity is just connecting things." Do you agree?

Pretty much. Though obviously it's about recognising which of all the connections you make is worth investing your time and energy in developing fully.

See : http://www.quora.com/Steve-Jobs-once-said-Creativity-is-just-connecting-things-Do-you-agree

Are there any companies out there that are as innovative as Apple?

Long term or short-term? In the slightly longer term I think Nintendo has a good track record.

Inventing formats : Game and Watch, NES, Gameboy, DS, Wii

High quality incremental developments of innovations : Super NES, Nintendo 64 etc.

Innovative content : Donkey Kong, Mario, Pokemon etc.

They don't always have hits. Sometimes another company comes along and dominates. But they're always exploring / pushing the boundaries. Finding their own take on things. And like Apple they have a good intuition about how to extract the synergies from their hardware / software / content mix.

See : http://www.quora.com/Design/Are-there-any-companies-out-there-that-are-as-innovative-as-Apple

Is it true that "customers don't know what they want until we've shown them," as Steve Jobs said?

Put it this way. Could you write your favourite novel? Play your favourite song? Cook your favourite meal? Genetically engineer the love of your life?

Most of us couldn't. Most of us don't have the capacity to deduce and produce all the things which will mean most to us.

Sometimes we have to let someone else, someone with more talent / practice / experience / specialization invent the things that delight us.

See : http://www.quora.com/Is-it-true-that-customers-dont-know-what-they-want-until-weve-shown-them-as-Steve-Jobs-said

When did the Googles of the world become more powerful innovators than its governments?

I think you have to unpack what you think “innovation” means. Not to mention “government”.

If innovation means certain large scale engineering projects then yes, government has been involved in everything from building viaducts to bring water to ancient Rome, to sewage systems in 19th century London, and sending people to the moon in the late 20th century. It’s a role it’s always had, and there’s no reason to think it won’t have again. Mega projects don’t happen every day, of course, so the odd 5, 20 or even 50 year gap between them probably signifies very little over this time-scale.

If you want a recent example of something government “did”, you could say the Higgs Boson was discovered by the government-funded CERN laboratory.

And yet there have also always been independent thinkers coming up with the best ideas.

Of course, as you go further into the past, it’s harder to demarcate what counts as “government” vs. “private” enterprise. Was Aristotle, the philosopher who developed a metaphysics and primitive scientific understanding that lasted a thousand years, an example of private enterprise? Or does the fact he was employed as a tutor to the Macedonian royal family mean he was part of the government?

Leonardo da Vinci designed war-machines for aristocrats. Again at a time when “aristocracy” was as much part of “government” machinery as any bureaucrat is today.

You already note this problem with respect to religion. Many great English scientists were officially employed by the Anglican Church, and the Anglican Church IS “established” (ie. part of the government in the UK.) But their research was a private matter, undirected by, and probably unknown to the government. They were “independents” for all practical purposes.

Since the nation-state took more of an active role in organizing education, most researchers get at least part of their education from state run or state funded institutions.

What about modern corporations? They’ve always done the “development” part of “research and development”, whether it’s innovations in steam engine design at the end of the 18th century or making the transistor and integrated circuit into commercially viable products.

Google Labs *sounds* exciting now. But not obviously *more* exciting than Bell Labs or Xerox Parc or even IBM Labs in their heyday. (Some of which was contemporaneous with the government funded moonshot.) At least since Edison, all big tech. corporations need to have a research lab cranking out cool stuff to amaze the public.

Silicon Valley is a Libertarian-ish kind of place. So it loves to tell stories about how Elon Musk is going to Mars and reinventing public transport before the stodgy old government. But this is all pretty speculative at this point. If government projects over-run 3 or 4 times their original cost projections, governments sometimes decide to live with that for strategic or vanity reasons. We’ll see if Elon Musk has the kind of patience and deep pockets to outspend the highly centralized, bureaucratic and corrupt Chinese government when it comes to going to Mars.

tl;dr : Innovation is a mixed ecosystem. With individual, free-lance geniuses, private initiative, corporate R&D, philanthropy AND government as funder of mega-projects. This pattern has survived thousands of years. Don't read too much into the minor short term fluctuations that are happening today.

See : http://www.quora.com/When-did-the-Googles-of-the-world-become-more-powerful-innovators-than-its-governments

Should we attribute the colossal failure of innovation in areas outside of computers to government regulation?

Firstly, what's your baseline for how much innovation there *ought* to be?

You have to know that before you can diagnose a colossal failure.

If your basis of comparison is the computer industry, then I'd say not. The computer industry has special characteristics. The main reason that the computer industry is so innovative is that software scales in a way that hardware doesn't.

A 2-person startup can launch a world-beating piece of software or a popular website. It doesn't need a staff that scales with the number of units sold. It may not even need much of a marketing budget if it's able to go viral.

There's really only one other industrial sector for which this is true : pop music. And we see similar fast innovation and trends there too.

Every other industry requires far more capital investment to turn an idea into a viable (let alone successful) product. That means that investors are the real gatekeepers to innovation. Far more influential than government regulation.

Now, the interesting thing is that the world is getting eaten by software. The culture and ideas of the software and web are spilling out everywhere else. Today it might be reasonable for a 2-person startup to try to design a new kind of car. Or a new food product etc. But they'll do it with the help of all the things software has given us : social networks instead of marketing. Kickstarter etc. for crowdfunding / pre-selling. New personal / small-scale tools like 3D printers / CNC routers etc. for the prototyping (largely a question of software). Huge amounts of knowledge available online.

So I'd say government regulation is more or less trivial compared to the characteristic of the market itself and when and how these tools become available.

The only place where regulation might have an effect is in medicine, where the government does place a greater restriction on bringing new products to market. OTOH how much do you want people dying of untested medicines or paying a fortune for medicines that don't actually work?

See : http://www.quora.com/Startups/Should-we-attribute-the-colossal-failure-of-innovation-in-areas-outside-of-computers-to-government-regulation

In the last 15 years, what are the most impressive innovations that happened in Silicon Valley that blew your mind?

15 years ago from 2013 is 1997.

Handhelds and mobiles aren't particularly shocking. We'd already had Apple's Newton. Go Corp. General Magic. Nah … nothing new there. The multi-touch screen is very sexy. That was an aesthetic delight. But not shocking.

Not even Google Glass. People were already going around with augmented reality, virtual reality goggles. You'd be reading about them in Wired every other month.

OTOH, the Google self-driving car is one of those things that I'd have dismissed as "yeah sure, but in practical terms it's just going to turn out to be too complicated to be feasible." It's the kind of thing we always joked was "20 years down the line" meaning that people were underestimating the difficulty.

Well, I'm genuinely shocked that they've actually been able to do it. That the computer power / algorithms are finally equal to the complexity of the traffic. Now, I think, we have to expect that the human-like androids ARE coming in the not too distant future.

See : http://www.quora.com/Silicon-Valley/In-the-last-15-years-what-are-the-most-impressive-innovations-that-happened-in-Silicon-Valley-that-blew-your-mind

When will technological development slow down?

Technological development is a function of information flow. Things which accelerate the sharing of ideas : open-cultures, bigger cities, the printing press, universities, the internet etc. tend to make technological development accelerate. Because it's easier for creative people to hear about problems that need solutions, and for good ideas to find good people to back them (with both finance and labour)

So the bad news (from your perspective) is that we're just at the beginning of an explosion of technological development, enabled by the internet, which is only going to get "worse" in the forseeable future as more and more of humanity gets connected, and the various cultures of open-sharing are embraced more fully.

Only three things might derail that :

- catastrophic ecological collapse (as climate change starts hitting the food-chain)
- catastrophic economic collapse (basically due to oil / energy shocks)
- the chilling effect of mass surveillance drives everyone away from public engagement and offline.

All three will be happening to some extent in the future, but it's an open question as to whether human ingenuity and, er, technoogical development, can come up with work-arounds to keep the whole show on the road.

See : http://www.quora.com/When-will-technological-development-slow-down

Are there any products or markets/industries where there is little to zero innovation taking place?

I'm not convinced toilet paper isn't evolving :

- the use of recycled paper or paper from sustainable forests to address eco-concerns
- luxury brands with extra padding / quilting
- toilet paper imbued with various kinds of scents
- novelty toilet paper with pictures printed on it
- or ordinary toilet paper with textured patterns

And then there's

There's plenty more scope. Perhaps toilet paper with the right agents added can be used for medical diagnostics (eg. to change colour in the presence of certain signs of disease)

At the end of the day, innovation is limited by human imagination. And there's no reason to think we've reached the end of that. Of course, there are some categories where the basic design patterns are pretty good and stable, so most innovation tends to be short-lived fads rather than something that is permanently adopted. But I don't think you can ever rule out the possibility that the NEXT innovation might be fundamental. In any category.

See : http://www.quora.com/Innovation/Are-there-any-products-or-markets-industries-where-there-is-little-to-zero-innovation-taking-place

Why is Silicon Valley not investing in healthcare as a sector other than pure healthcare IT plays?

a) Healthcare has a lot of regulation. Many SV entrepreneurs don't like / understand regulation or want to have to spend their time dealing with lawyers rather than code / customers.

Furthermore, many products can't be rolled out without clinical trials which can take years. This is a barrier for startup culture : funding doesn't last that long; entrepreneurs are often, by nature, restless and impatient.

b) SV entrepreneurs tend to make products for people like themselves or their friends. That's not an entirely stupid heuristic for young, inexperienced entrepreneurs to have. At least they'll know something about the customers.

But the main customer base for health-care products are the old, ill, infirm, poor, chronically fatigued etc. People that the healthy young things of SV don't really relate to. (Contrast with *fitness* products which healthy young people and SV does love.)

See : http://www.quora.com/Why-is-Silicon-Valley-not-investing-in-healthcare-as-a-sector-other-than-pure-healthcare-IT-plays

Do African startups stand a chance against well established companies like Google, Facebook and Twitter?

```
        It's very rare for a new startup, anywhere in the world, to directly cha\
llenge the incumbents head on. The way a small startup becomes successful is to \
disrupt the market.
```

Disruption has a very specific meaning. It means finding a nascent market that the incumbents DON'T currently serve well, and serving that while the market itself grows. Microsoft didn't challenge IBM in making mainframes or typewriters. It did something that was initially too small for IBM to care about : PC operating systems. It grew on the back of that market. It only came into direct conflict with IBM about a decade into its existence, when IBM realized that it too wanted to be in the (now huge) desktop PC operating system market and came out with O/S 2.

Similarly Google started in areas that Microsoft didn't care about (partly search and partly micro-transactions for adverts) and didn't come into direct conflict for several years until Microsoft realized they wanted to be a search giant and Google decided they could make better (more focused) laptop operating systems than Windows

An African copy of Google, Facebook or Twitter won't go anywhere. But neither will a new copy of Google, Facebook or Twitter from Silicon Valley. An African startup has every chance if it picks the right large, underserved market that the current giants can't be bothered with and grows with that market.

I don't know Africa. By all accounts it has several fast growing economies. A relatively high penetration of mobile phone use and sophistication of mobile phone users (people are more used to doing research and making financial transactions on their phones than Europeans or North Americans). The Chinese are making large investments in land and mining there. Who knows what that confluence of factors combined with Africa's varied cultures and histories will bring. I'd suggest that an African startup has more or less the same chance as anyone else in inventing the next world-beating augmented reality app. or an interesting peer-lending scheme or some crucial B2B service that hooks into the APIs of Chinese social networks. Of course, African startups are going to face the problems of raising money from Silicon Valley VCs. And a cost of promoting their startup in the US. But these days there are ways you can grow without VCs, and the online world is a lot bigger than the US. ## How can I start a space exploration company?

You start the company like any other.

Your main problem is that most things in space are expensive and unless you are already very rich or have an idea and team that are VERY compelling to investors, then you probably can't afford to do much in this area, EXCEPT process data which other people are generating.

Google are clever in that their current proposed project involves putting up some mediumly expensive telescopes to scan asteroids but largely their contribution is ground-based data-processing.

The cheapest way to get somewhere in space is a project like (which does seem to have launched now, would be interesting to know how it's going and whether there are similar plans in the near future.) If you can think of commercial opportunity with that kind of satellite, then something along those lines seems reasonable.

Your best bet, though is to identify interesting opportunities in processing data to look for something that other people currently aren't and which might have commercial value.

See : http://www.quora.com/Space-Exploration-1/How-can-I-start-a-space-exploration-company

Social Media

What exactly is the Internet?

It's a communication protocol that includes :

- an addressing system for computers
- a routing algorithm for how to get messages from one computer to another

That's basically it.

There are different low-level protocols for how individual computers can talk to each other (including one over) but IP (internet protocol) doesn't care which you use. And there are a bunch of other protocols for sending specific kinds of data which run on top of the internet. (Most famously http for sending web pages and other files around)

But the internet itself is just that protocol for how computers are addressed and should route information.

See : http://www.quora.com/What-exactly-is-the-Internet

Why should (or shouldn't) you migrate your blog to Quora?

The internet is a two class society :

- those who own their own domain names (and have their assets located at those names) are first-class citizens. They are free men and women. At any time they can up-sticks, take their identity and content away from the current host and put it somewhere else. Even if it's a certain amount of work to extract and reformat it, they can do that without losing their address and audience.
- those who don't own their domain names and park their assets at someone else's domain (Facebook, Blogger, Quora) are basically sharecroppers or feudal serfs. They're owned by the lords of the particular manor which they've attached themselves to.

The longer you wait to get your own domain, the longer you are in servitude. No service or software, whatever the functionality is worth giving up your own domain for. And if you don't have a domain, but are planning to make the effort to break from your existing owner, don't just walk straight into another captivity.

See : http://www.quora.com/Blogs-on-Quora/Why-should-or-shouldnt-you-migrate-your-blog-to-Quora

Consumer web is highly fragmented. How will this trend continue in the next 5-10 years, and what will the web look like then?

The really big web-trend of the future is ubiquitous computing. The proliferation of different devices : from phones and tablet to glasses, watches, windscreens to drones and other robots. In 10 years time, the web will be everywhere in our lives and artefacts.

These devices will offer widely diverse ways of accepting *input* and engaging the world - making it impossible to have common applications across them.

Software (including web "pages" / apps) will become a lot more pro-active : sensing what's going on around it and making intelligent interventions in the world.

As to "web" technologies themselves. HTTP will be increasingly supplimented by other web-socket based connection protocols. I expect either a) HTML becomes far less important. Or b) people to invent similar markup-languages for other kinds of application such as to define patterns of robot behaviour.

OTOH, the javascript virtual machines (V8, Gecko) etc. will become more important. People will increasingly experiment with new languages and ideas on top of javascript. Declarative / constraint-based / reactive programming techniques will become increasingly important. (Ideas you see in angular.js, elm-lang, constraintsjs, meteor.js etc.) I'd especially predict declarative languages to define sensor inputs and an event model to handle it.

See : http://www.quora.com/The-Future/Consumer-web-is-highly-fragmented-How-will-this-trend-continue-in-the-next-5-10-years-and-what-will-the-web-look-like-then

Why hasn't anyone disrupted LinkedIn yet?

Partly because no-one knows what LinkedIn actually DOES. So it's hard to know how to do it better. :-)

I used to say LinkedIn's strength was that it wasn't demanding. You just parked your CV there and so did everyone else. And that's it. You don't have to go back to it every day or even every week. And that was fine. There's a need for a site like that. Being the trusted place to keep up-to-date resumes and contacts was its niche.

Now it's woken up and decided it DOES want to be active and try to get me back there every day. It tries to push "news" at me. It looks more like Facebook. It has "discussions".

But none of this is particularly good. Quora does Q&A and discussion better. Facebook is a better Facebook. Twitter is a better Twitter. The newsy fragments are risible.

So it's starting to look increasingly devoid of any real vision or road-map to providing extra value.

The area that LinkedIn *should* be expanding in is in helping us manage more sophisticated "portfolio" styles of working. It should at least be looking at what's going at oDesk or (extreme example) Fiverr. Short term contracts are the future of work, so what if I have 50 jobs a year? But they all lasted a couple of days? How does my CV look then? What can LinkedIn do to help me find the next 10 or 20 gigs during the next two months?

What else should LinkedIn be telling me about how work is evolving and how should it be helping me increase my income? This seems a mission worthy of a site that aspires to be an internet giant like Facebook or Google. But I have little hope of seeing them rise to the challenge. I think scrappy little companies like Fiverr and About.me and even Behance (now part of Adobe) are leading the way here. (And as Jon Bischke (http://www.quora.com/Jon-Bischke) reminds us, all those sites like GitHub / StackExchange that are part of our professional technical identities.)

See : http://www.quora.com/LinkedIn-6/Why-hasnt-anyone-disrupted-LinkedIn-yet

Would you want to be part of a LinkedIn Most Connected Person's network?

Not really. What's important on LInkedIn is the value of the links. Ie. If I know X and X knows Y can X help put me in touch with Y / put in a good word for me etc?

If X is someone I don't really know, but just collects as many connections as possible, it's not likely that those connections will be very meaningful to anyone else. If I ask X to connect me to his "friend" Y it's, frankly, not that likely that Y will pay much attention.

See : http://www.quora.com/LinkedIn-6/Would-you-want-to-be-part-of-a-LinkedIn-Most-Connected-Persons-network

Do you accept invitations to connect from people you haven't met, let alone done business with?

I do if I've known them in some way online. Eg. if I follow their blog or regularly see them in a forum. Otherwise no. I don't think I've ever accepted a request from a complete stranger simply because their profile looked cool.

See : http://www.quora.com/LinkedIn-6/Do-you-accept-invitations-to-connect-from-people-you-havent-met-let-alone-done-business-with

As more content on the web is published in the form of data, are monolithic search engines still useful?

Although the quantity of machine-to-machine data on the web will expand and potentially swamp the human-to-human data, it's the human written, human readable data that humans are really interested in.

So there's always going to need to be a human-readable interface to search human-readable documents, and I don't see much reason to think that that requirement will scale faster than the monolithic search-engines' attempts to keep up.

See : http://www.quora.com/As-more-content-on-the-web-is-published-in-the-form-of-data-are-monolithic-search-engines-still-useful

How does Facebook know I am gay?

I have no insider knowledge, but here are some plausible hypotheses :

1) you've visited gay sites that have Facebook "like" buttons on them. Every time one of these pages loaded, it called back to the Facebook server to get that button. The Facebook server then got to look at the cookies that it left on your machine when you were using Facebook itself, so it could identify who you were.

Therefore Facebook knows that you visit those sites, even though you've never actually clicked on any of those "like" buttons. Just the fact that the buttons are on the page allows it to put 2+2 together.

2) you have a lot of gay friends on Facebook who do list their sexuality.

Yes, this is a scandal. Everyone should understand this about Facebook (and other similar social sites) Everyone should understand this part of how the web works. If someone can put a button on a page, that someone gets to see when you access that page. And can tie your movements to all the other pages with their buttons which you've been to (which for Facebook is a lot)

Facebook knows more about you (and more about most of its users) than almost anyone else does.

If you want to protect yourself, get a cookie blocker / ad-blocker plugin for your browser. Although, ideally, you should stop using Facebook if you can. (I closed my account last year. Partly because of the intrusive surveillance culture that sites like FB are growing into.)

Update : it's worth reading this story : Facebook turns user tracking 'bug' into data mining 'feature' for advertisers (http://www.zdnet.com/facebook-turns-user-tracking-bug-into-data-mining-feature-for-advertisers-7000030603/?s_cid=e539&ttag=e539&ftag=TRE17cfd61)

See : http://www.quora.com/Facebook-company/How-does-Facebook-know-I-am-gay

Who has visited Join4Likes.com-Get More Free Likes, Shares, Twitts, Followers and more? I think it is a very nice tool to increase your social media presence.

What's the point of a bot which automatically clicks a button that says it "likes" you?

What you want is real people to like you. The kind of likes you get from such tools and services are either just machines. Or people who have been duped into liking because they think they're doing something else. Or because they're hoping to earn "likes" of their own.

It's desperate and pointless. You're fooling yourself, artificially boosting your "popularity" with fake approval.

See : http://www.quora.com/Who-has-visited-Join4Likes-com-Get-More-Free-Likes-Shares-Twitts-Followers-and-more-I-think-it-is-a-very-nice-tool-to-increase-your-social-media-presence

You ever think of buying Plays on Soundcloud for a kick start?

Strikes me as idiotic.

Attention which is for sale in that way is basically the attention of bots, not people. It's either a bot which has compromised a user machine or account which is doing the following / listening. Or it's a bot which has followed thousands of people and which a bunch of naive (probably not very serious) users have followed back as a reflex action. The sort of people who unthinkingly follow bot accounts are very unlikely to be discerning or valuable listeners to your music.

If you want to pay money to get your music listened to it's probably better to invest it in either buying ordinary advertising (ad-words, a fragment in a printed music fan-zine which at least has engaged music fans reading it) or buying time with a producer / mastering studio to help make your music sound more professional.

See : http://www.quora.com/Music/You-ever-think-of-buying-Plays-on-Soundcloud-for-a-kick-start

What is the differece between blogger and google site builder?

I haven't looked at Google Site Builder for a long time. I'm not even sure if it's still supported.

Traditionally the difference was this :

- Blogger is (obviously) blogging software, that came from the culture of blogging and emphasized the tools that bloggers needed. It's by no means as full featured as WordPress, but it does most of the basics, reasonably well.

- Google Site Builder was for complete novices on the web (typically small businesses) to put up some kind of web-presence. It's main focus was to be as easy as possible for someone with no knowledge to make some kind of page with their other contact details (phone, address) without having to pay a web-developer to make one for you.

What's interesting is where this is going in future.

No-one should underestimate how much the *thinking* about the web has *changed* since the rise of Facebook and Twitter. Today, most people's basic "web-presence" is their Facebook page. There are people who still have no idea about what a website is, or how (or why) they could create one, but still use Facebook everyday as an app. on their mobile device.

Some people use Twitter similarly.

Now it used to be *obvious* to most of us in web-culture that people should have some kind of web-page. And things like blogs and site-builder type sites were ways to get that. Today there's an assumption that FB or Twitter might be all that most of humanity need.

Now, to be very clear I don't agree with this. I think it's an extremely important principle that people should own their own domain names and have some kind of site of their own. If you DON'T own your own domain name you are effectively a serf online, entirely dependent on and exploited by the corporation you've attached yourself to. (BTW : I'm even starting to run open-events / workshops under a "reclaim your domain" banner in the town I live in, to help people understand this principle and get their domain name.)

But to stand back from the political for a minute, the large corporations, the internet industry today, is now focused on getting people to sign-up and be locked-in to Facebook-like things : accounts that have statuses and followers all within the site itself.

So here's what I expect to happen. I don't think Google, right now, care very much about site-builder. (I have a no inside knowledge, possibly they have a strategic use for it, but haven't heard of one.)

Meanwhile Google's longer-term strategy (I believe) will be to merge Blogger into GooglePlus (their Facebook-like thing). They already merged the comments systems into more general G+ comments. (If you enable this on Blogger, only G+ users can comment on your posts.) They have a way to automatically link your new Blogger posts on G+. I think next time they redesign Blogger it will be even more integrated with and like G+ .

If I'm right about that, don't expect Blogger to start getting any tools that are not "Facebook-like" in some sense. So it might get better tools for creating events (that's something that Facebook lets individuals do), but it's unlikely to give you the ability to run a shop.

See : http://www.quora.com/What-is-the-differece-between-blogger-and-google-site-builder

Will Facebook suffer the same fate that MySpace did? Why or why not?

No. Mark Zuckerberg, for all his faults, understands social media infinitely better than Rupert Murdoch and his lackeys.

See : http://www.quora.com/Facebook-product/Will-Facebook-suffer-the-same-fate-that-MySpace-did-Why-or-why-not

What are the technological and cultural legacies of MUDs?

Also, all the MMORPGs from Ultima Online, World of Warcraft, Diablo, Everquest etc.

See : http://www.quora.com/What-are-the-technological-and-cultural-legacies-of-MUDs

How can we create a global community for sharing ACTIONABLE IDEAS in any field of endeavour?

There have been various attempts to build things like this. The main problem is that votes on a site don't really signal any kind of commitment from the people doing the voting.

Everyone can *say* they want someone to do something about blah blah, but if that isn't backed by some more concrete commitment (money / time) then it means very little. It doesn't help allocate scarce resources (because people can "like" far more than there are resources to satisfy) and it doesn't bring more resources to the table.

"Actionable Ideas" are only really "actionable" if someone is ready to put them into action. It's not a property of the idea itself. It's a relational property between the idea and the *actor.*

So, to me, the most promising examples of working systems like this are Kickstarter and Indiegogo etc. Platforms where people put up their ideas in the form of well defined projects / products with an already assembled team committed to carrying them out (if they get funding). And where "votes" come in the form of cash commitments.

This doesn't have to be just about preselling videogames or gizmos. You can do a lot of good in this format (eg.) I think that these kinds of site are truly revolutionary, and can plausibly bring us exactly the kinds of benefits you're hoping for.

See : http://www.quora.com/Open-Source/How-can-we-create-a-global-community-for-sharing-ACTIONABLE-IDEAS-in-any-field-of-endeavour

Would you use a Twitter client that sorts the tweets based on their relevance, instead of timeline?

Given how short Tweets are it seems that "relevance" is going to be hard to make sense of. Maybe some links may be more relevant than others but relevant to what context?

What I'm interested in when I happen to log into Twitter? How would you or Twitter know?

To my last Tweet? That's hard to figure out.

If PEOPLE aren't interesting to me, I don't follow them. If there's an event / particular subject, hashtags do a reasonable job of focusing on them. I'm not sure I believe you'll get a significant improvement over those two mechanisms.

See : http://www.quora.com/Would-you-use-a-Twitter-client-that-sorts-the-tweets-based-on-their-relevance-instead-of-timeline

Robotics

What is your opinion on the three laws of robotics?

It's an interesting exercise in philosophical fantasy to come up with a mechanizable ethics. To try to imagine ethics as an algorithm. And obviously it's fascinating to see all the ways that such systems fail or fall into paradoxes.

In future we may very well face situations where we do want to mechanize ethics and we'll almost certainly find ourselves with similar problems. In fact we should probably be doing it already. Eg. the kind of NSA mass analysis of communications must have some "laws" built into it, to decide who is considered legitimate target and who isn't. Future automated policing systems, in banks, in streets etc. will almost certainly be making analyses of the ethics of the people they're observing.

As Asimov himself demonstrated, the laws themselves are clearly insufficient though they aren't a bad first stab at a robot-ethics.

See : http://www.quora.com/Three-Laws-of-Robotics/What-is-your-opinion-on-the-three-laws-of-robotics

How will robots and robotics be a part of Google's future? What products and services will Google offer?

Google are fortunate enough that they still have money to play /explore without having a fixed game-plan. I'm pretty sure that when they picked up Android, the thinking wasn't : "what services will we have around this?" It was "how can we ensure we have levers of control in the coming mobile era?"

I expect it's the same here. They know enough, now, after the self-driving car etc, to see that the robots are finally here : the engineering is sufficiently slick, the computers are powerful enough, the 4G is fast enough that crunching Big Data in the cloud is an option. So how do they ensure their dominant position in the coming robot revolution?

See : http://www.quora.com/Googles-Robotics-Strategy/How-will-robots-and-robotics-be-a-part-of-Googles-future-What-products-and-services-will-Google-offer

Will robots make people jobless?

Yes.

Automation usually does. That's what it's for.

The question is, can the economy and participants evolve faster than technology, in order to create new jobs and to learn to do them?

That's an open question.

One school of thought says "we always did in the past, we will in the future". That's what I call the "naive inductionist" argument.

It might be right. But it's based on nothing but naive induction all the same.

Another argument would be to say, the economy / population have a certain maximum rate of adaptation / learning and when the rate of technological change overtakes that, we're in trouble.

One thing we can be sure of is that technological change is cumulative and accelerating. So we're in trouble. If not yet, sometime in the future.

The third position might be that economic evolution and learning is also accelerated by the same forces that accelerate productivity. Maybe social networks, Kickstarter, etc. will accelerate our capacity to dream, and invent new desires and wants, hence speeding up the creation of new jobs.

Right now we have very little good data or theory for this argument. We have anecdotal evidence. I don't believe, so far, that we have many models that can be tested against reality. So believing it is an act more of blind optimism than sound reasoning.

See : http://www.quora.com/Robotics/Will-robots-make-people-jobless

What will be the next big advancement in robotics within the next 10 years?

All of them :-)

Seriously, robotics is exploding now …

- more powerful, cheap, low powered processors (thanks to the mass market for mobiles and the "internet of things")
- lots of cheap, powerful sensors (thanks to the "internet of things")
- several layers of communication system (eg. Google and others have built out the cloud to receive data from mobile devices - including robots - to do offline processing, informed by other data … eg. Google's self-driving cars can get map information, while both receiving and contibuting to up-to-date traffic information etc.)
- new generations of robotics hobbyists (who cut their teeth in the Arduino / maker communities) starting to invent and publish their ideas, launch startups and Kickstarter projects etc. for circuit boards, drones or other complete robots.

I’m pretty sure we’ll have robots that operate safely, smoothly and effectively in natural human environments in the next 10 years. They may not be humanoid, they may fly or roll on wheels. But they’ll be interacting with us.

See : http://www.quora.com/What-will-be-the-next-big-advancement-in-robotics-within-the-next-10-years

What kind of baby care tasks would you be willing to delegate, at least partially, to a robot?

Robots already do baby / child-caring.

Televisions keep children occupied while busy mothers do other things.

Increasingly robotic toys entertain them. And sometimes teach them.

It’s very plausible that in the near future there’ll be some way to put a tag on your baby / child and have an app. on your iPhone monitoring it. People will think this is weird right up until there’s a sea-change in social mores, after which and it will be considered irresponsible for parents NOT to do this.

We have child-proof locks on cars. It’s not that hard to imagine a future with child-proof locks on apartments, allowing the parents to pop out to the shops (or for a couple of hours work). The apartment can phone if there’s a problem.

I think we’re going to get Philip K. Dick’s “Electric Sheep”. Increasingly sophisticated automated toy animals and dolls with which children will have increasingly long and sophisticated interactions with. Can we imagine a point where you can give a real (robotic) Hobbes to your young Calvin? I don’t think that’s as far off as people may imagine.

See : http://www.quora.com/Business/What-kind-of-baby-care-tasks-would-you-be-willing-to-delegate-at-least-partially-to-a-robot

Will professors be replaced by robots?

Define “robot”.

Online video of professors and educational animation replace live performance by professors.

Online discussion forums and MOOCs replace the seminar chaired by a professor (or grad student)

Some marking can be automated (of multi-choice questions or code).

Marking of essays still requires skilled humans. But can perhaps be outsourced to people who never held this job before. For example, people who have Master’s degrees and could, theoretically be doing a PhD, but aren’t due to financial constraints, bringing up children etc.

“Smart” Text-Book apps. can, perhaps, offer more interactive learning experiences which would have previously required guidance from a professor.

So automation will certainly replace (a large proportion of) professors. But not necessarily packaged into a single robotic body.

See : http://www.quora.com/Will-professors-be-replaced-by-robots

Would you ever buy clothing from a vending machine?

People buy clothes on the internet all the time. That already dispenses with the idea that you need to see / feel / try-on before buying.

Whether a vending machine with its necessarily limited selection can compete with the internet’s other advantages (huge range, low price) is another matter.

See : http://www.quora.com/Vending-Machines/Would-you-ever-buy-clothing-from-a-vending-machine

If I someday want to build a battle bot or similar robot, where do I begin?

Get an Arduino and robotics platform (eg … It’s just a simple vehicle you can learn to drive around but it’s cheap, there’s lots of documentation and online advice)

Then move up to something like a RaspberryPi / BeagleBone Black and use your construction skills to build a larger, more powerful body.

See : http://www.quora.com/Robotics/If-I-someday-want-to-build-a-battle-bot-or-similar-robot-where-do-I-begin

When will robot bees be able to replace bees for flower pollination and honey production?

Bloody expensive, considering how cheap and distributed bees are. You’d have to get the bees down to a couple of cents each to compete with bees.

Either that or drive the bees extinct first.

See : http://www.quora.com/Robotics/When-will-robot-bees-be-able-to-replace-bees-for-flower-pollination-and-honey-production

Can a software engineer get a job programming robots?

Yes. But it probably requires there to be some kind of robot industry in the region where you want to live and work.

It's probably harder for robot programmers to work remotely than for a web-designer because you'll need to have a physical robot in front of you, to really understand what your code is doing.

The bigger question then, might be, how easy is it for a region which has a some good software engineers to start getting into robotics? Do you need to have a history of mechatronic engineering too, to have some hope of that?

See : http://www.quora.com/Software-Engineering/Can-a-software-engineer-get-a-job-programming-robots

Does any programming language have "when (expression) do {block}"?

Urbiscript is a wonderful language for programming robots where pretty much everything is organized around this reactive paradigm :

urbiscript (https://en.wikipedia.org/wiki/Urbiscript)

Concurrency is a built in primitive and event handlers are all reacting concurrently.

See : http://www.quora.com/Does-any-programming-language-have-when-expression-do-block

Micro Markets

What can micro-gig marketplaces like fiverr and taskrabbit do to upsell their users?

I think Fiverr is doing the right thing by allowing sellers to offer higher-value / higher-priced add-ons to the basic $5 gig.

Could it go further? I'd guess once it starts building up better feedback about vendors it could use that in some way. Give badges / credits to successful / reliable vendors. Allow privileged vendors to create gigs with a minimum of $25 or $50.

A bolder move might be to try offering a Kickstarter-like service. Allow gig vendors to create Kickstarter-like projects but at a smaller-scale than the average Kickstarter. For example "I will write this short ebook if 20 people commit to buying it for $5". Or "I'll cycle naked through Manhattan if 500 people pay $10."

Or look at something like CafePress which has been a great idea for over a decade but feels tired. Fiverr is full of creative artists and performers. Many of whom are using it to build their personal brand. Maybe those vendors would love the option of offering a t-shirt or other branded items to happy customers etc. right on their page.

Fiverr could also beef up the vendor profiles in other ways. Steal good ideas from everywhere from Behance to LinkedIn to DesignOutpost to oDesk to vizify.

See : http://www.quora.com/Marketplace-Startups-and-Companies/What-can-micro-gig-marketplaces-like-fiverr-and-taskrabbit-do-to-upsell-their-users

Does anyone use Fiverr?

I've used it a few times.

I've had small bits of HTML / CSS done. And some fairly standard icons designed. I also had a tune I wrote mastered.

In summary, I think all the services I received were remarkably good / good value for $5. None of them were perfect. (The design-work isn't perfect because no-one can read my mind, and although I asked for a couple of iterations in some cases, I'm too embarrassed to keep pushing it until it's really right, given the low price I'm paying.)

If you go there expecting you'll get the same service / quality as you'd normally have to pay 100 or 1000 times for from a local supplier, then you'll be out of luck. (And frankly you deserve to be.)

If you use it as a useful way to get rough / quick prototypes / ideas / sketches knocked up that you plan to polish yourself later, then it can be a useful part of your workflow.

See : http://www.quora.com/Fiverr/Does-anyone-use-Fiverr