

Introduction to Programming

Ben Evans

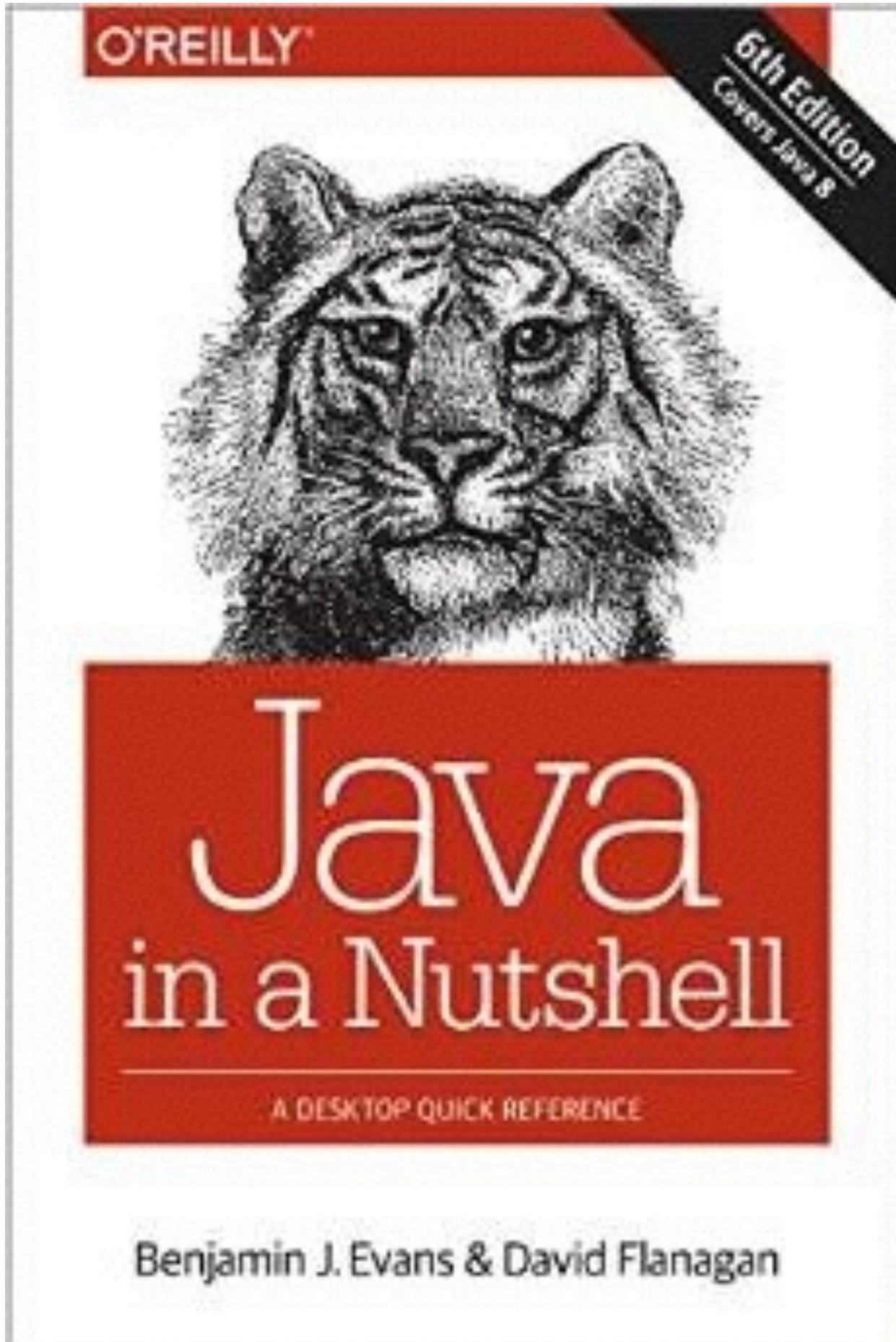
About Me

- Co-founder & Tech Fellow, jClarity
- Java Champion & JavaOne Rock Star Speaker
- Java Community Process Executive Committee
- London Java Community
 - Organising Team, Co-founder, AdoptAJSR & AdoptOpenJDK



About Me

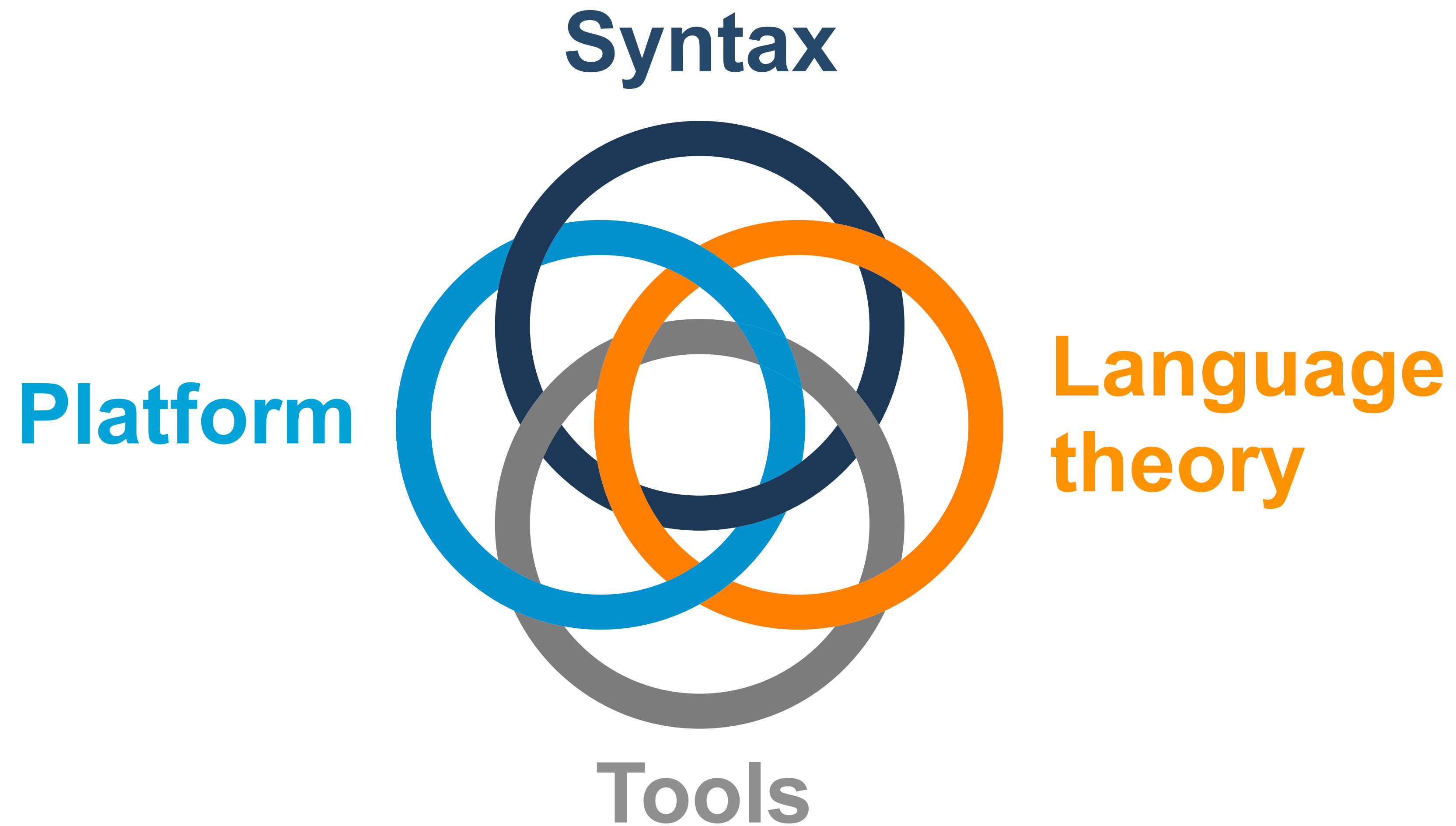
- Java in a Nutshell (6th Edition)
- Introduction to Java 8
- The Well-Grounded Java Developer
- Previously:
 - Deutsche Bank - Chief Architect (Listed Derivatives)
 - Morgan Stanley - MATRIX, Google IPO, Dodd-Frank
 - SportingBet - Chief Architect
 - Researcher - Quark-Gluon Plasma Physics



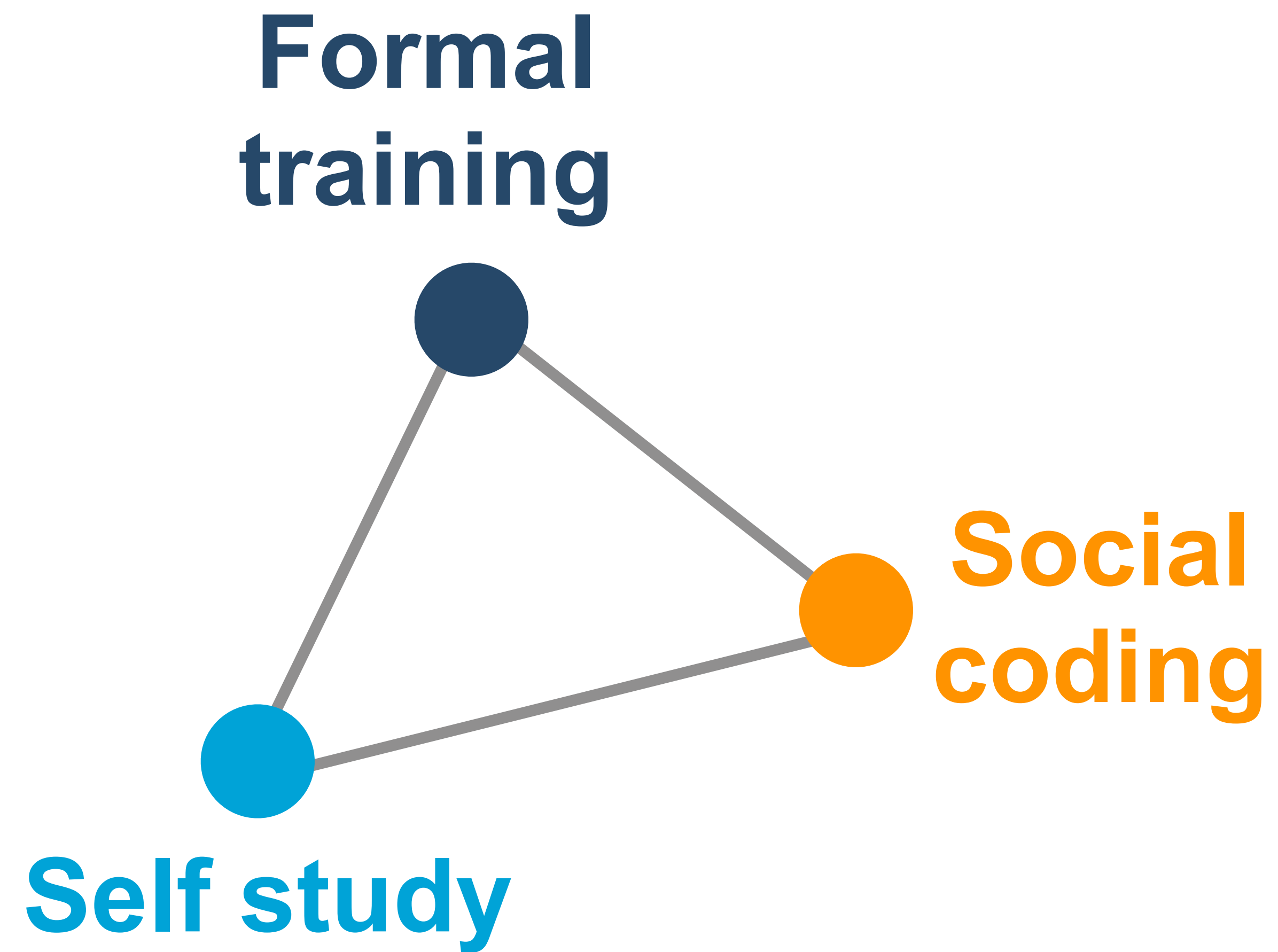
How Do We Learn?



Why is Programming Different?



How Do We Learn?



Operating Systems & Linux

- Introduction
- Some History
- Basic Commands
- UNIX Philosophy
- The Pipeline

Introduction

- Why do we need operating systems?
 - Control access to scarce resources
 - CPU time
 - I/O Resources
 - Memory

OS Components

- Operating system kernel
- Scheduler
- Virtual Memory Subsystem
- Standard library & kernel interface

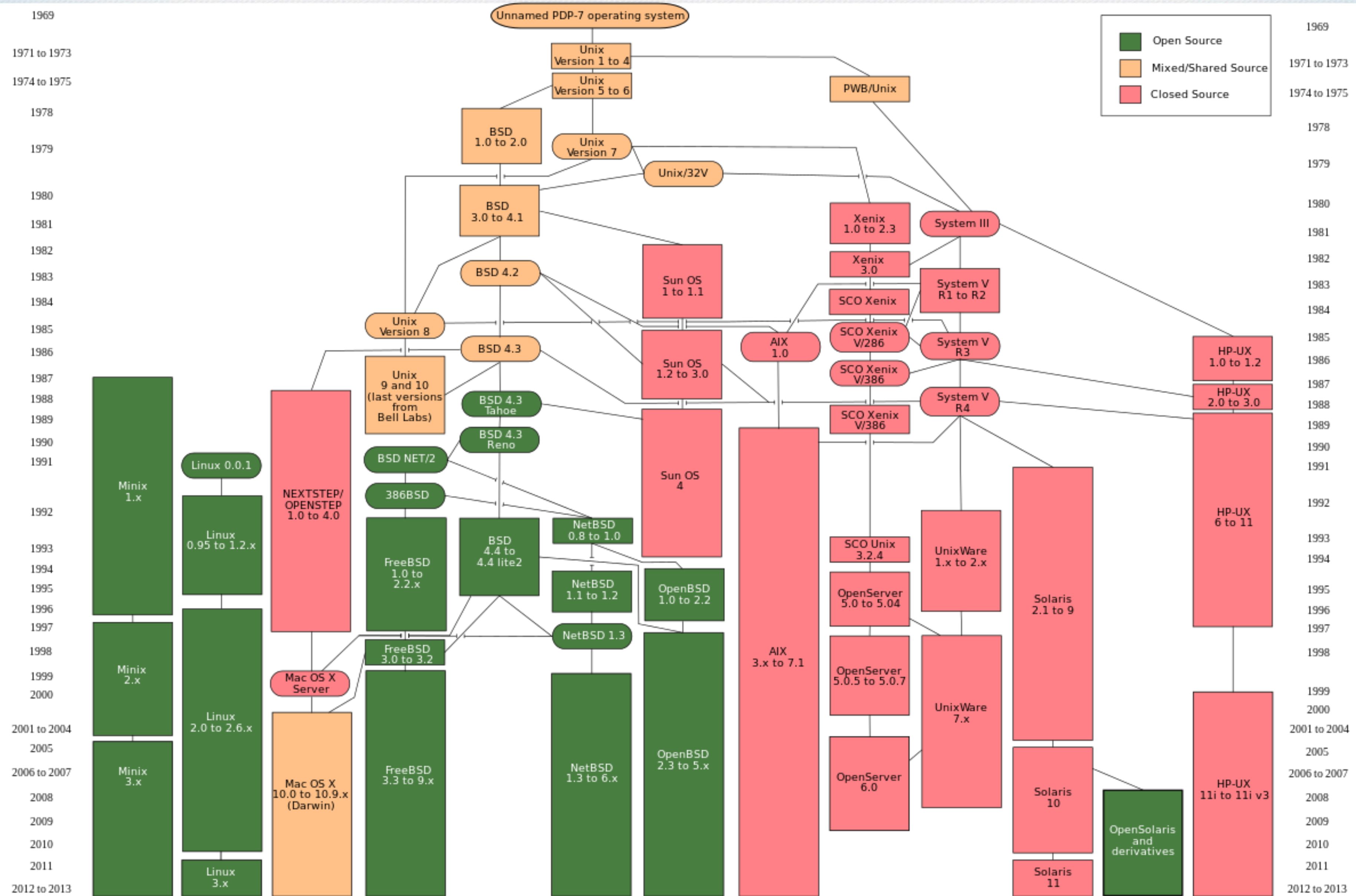
Some History

- Why Unix?
- The 80s & 90s
- The Rise of Linux & Decline of Solaris
- Modern Times

The beginning

- Unix was originally developed in 1970s
 - Along with the C programming language
 - Dennis Ritchie & Ken Thompson
- Quickly became very popular
 - Many variants and different versions
- A “family” of related operating systems
 - Mostly standardised (but some differences)

A family tree of Unix



The 80s & 90s

- BSD Unix
- Sun Microsystems (SunOS & Solaris)
- IBM AIX
- Silicon Graphics
- Digital Unix
- Could Windows Challenge?

The Rise of Linux & Decline of other Unixes

- Linus Torvalds released first version in October 1991
- Quickly became dominant open OS for servers
 - BSDs held up by court battles
- Sun's Solaris still led in the data centre
- First Dotcom crash
 - Wave of cost-cutting (Linux seen as cheaper)

Modern Times

- Linux is everywhere
 - Android
- Macs provide a different Unix variant
 - BSD-based
- Solaris has a special niche (Oracle acquisition)
 - But OpenSolaris forks (e.g. CoreOS) continue

Basic Commands

- The Terminal
- Getting help
- Files & the Filesystem
- Processes
- The shell

The Terminal

- Linux Users - Assuming you can get there!
- Mac Users - In Finder:
 - Applications -> Utilities -> Terminal
- Windows Users - Start up VirtualBox

Exercise 0

- Get the Terminal started up
 - Mac Users may want to add the Terminal app to the Dock
 - Windows users need to get VirtualBox started
 - Linux users - Coffee Time!

Getting Help

- man - “Manual Page”
- Looking for help online
 - Stack Overflow
 - Learning the terminology (help guide search results)

Unix Command Format

- Linux commands have “switches” (aka options)
 - Extra optional functionality
- General format of a command:
 - `<COMMAND> <SWITCHES> <ARGUMENTS>`

Working With Files

- `ls` - “List files”
- `file` - “Examine type of file”
- `wc` - “Word Count”
- `cp` - “Copy File”
- `mv` - “Move File”
- `rm` - “Delete File”

Working With Files

- cat - “Display contents of file”
- head - “First few lines of file”
- tail - “Last few lines of file”
- less - “Display file a page at a time”

Lab 1 - Basic File Manipulation

1. List the files in the directory
2. What types are they all?
3. Display the contents of one of them
4. Use the “man” command to explore the switches to the file listing command
5. How can you get a “long listing” of all the details of a particular file?
6. There are three “hidden” files in the directory - what are they?
7. How many words are in the file called south.txt?
8. Can you rename south.txt to north.txt?

Useful Switches

- `ls` - “List files”
 - `-l` : Long Listing
 - `-a` : All files (show hidden)
 - `-h` : Human Readable file size
 - `-t` : Sort by when file was modified
 - `-r` : Reverse order (useful with `-t`)
 - Can combine: `ls -lh` or `ls -la`

Useful Switches

- `wc` - “Word Count”
 - `-l` : “Line count”
 - `-w` : Per file count (if multiple files given)
- `rm` - “Delete File”
 - `-f` : “Force” (sometimes useful)
 - `-r` : Recursive - delete subdirectories & contents (CAREFUL)
 - `-v` : Verbose (show the name of each file as it’s deleted)

Useful Switches

- head - “First few lines of file”
 - -n : Display the first n lines of a file
- tail - “Last few lines of file”
 - -n : Display the last n lines of a file
- less - “Display file a page at a time”
 - +n : Skip past first n lines of file before starting display

Working With Directories

- pwd - “Print Working Directory”
- cd - “Change Directory”
- mkdir - “Make directory”
- rmdir - “Remove directory”
- df - “Disk Free?”
- du - “Disk Used?”

Files & Filesystems

Desktop

Applications

Documents

Bank

College

Insurance

Personal

Travel

Writing

Desktop

Dissertation

Guidance.pdf

Proposals

QMUL stats

Year review

_Updated list

Archived

D3837notes.pdf


D3837 v1.key

Feedback notes

M9273.pdf

M9273.pdf

PYTH-628.pdf



Name:

M9273.pdf

Type: PDF

Size: 2.3Mb

Created:

21-05-2015

Open with:

Acrobat Reader

Find

- find - find files
- Recursive directory traversal
 - -iname : find by name
 - Many, many other switches..
- Consult the man page

Working With Processes

- `ps` - “List processes”
 - `ps -ef` (`ps auxw` on Mac) - show details
- `kill` - “End a process”
 - Takes a process ID

The Unix Shell

- A fundamental piece of Unix
- A text-based way to interact with the OS
 - Pre-dates GUIs
 - Very powerful (in the hands of an experienced user)
- Input commands & receive output
- When you start a terminal, it's really the shell that starts

Unix Philosophy

- Files and processes
- The Unix Way
- How does the philosophy help?

Files and Processes

- Unix has a minimum set of abstractions
 - Files
 - Processes
- “Everything is a file”
 - Resources are treated as simple streams of bytes
 - Same tools can be reused across resources

The Unix Way

- Write programs that do one thing and do it well
- Write programs that expect to work together
- Write programs to handle text streams
 - Because that is a universal interface
- Design & build software to be tried early
- Data structures, not algorithms are central

The Unix Way

- Modularity: Write simple parts connected by clean interfaces.
- Clarity: Clarity is better than cleverness.
- Composition: Design programs to be connected to other programs.
- Simplicity: Design for simplicity; add complexity only where you must.
- Transparency: Design for visibility to make inspection and debugging easier.
- Silence: When a program has nothing surprising to say, it should say nothing

The Unix Way

- Repair: When you must fail, fail noisily and as soon as possible
- Economy: Programmer time is expensive; conserve it in preference to machine time
- Optimization: Prototype before polish. Make it work before optimizing
- Extensibility: Design for the future. It will be here sooner than you think
- Least Surprise: In interface design, always do least surprising thing

The Pipeline & Linux Programming

- STDIN & STDOUT
- grep
- Regular Expressions
- The Pipeline
- Tools for the Pipeline

Unix Pipelines

- STDIN : Standard Input (<)
- STDOUT : Standard Output (>)
- STDERR : Standard Error (2>)

```
boxcat$ wc south.txt
14233  150375  843763 south.txt
boxcat$ wc south.txt > south-word-count.txt
boxcat$ cat south-word-count.txt
14233  150375  843763 south.txt
boxcat$
```

grep

- grep - “Look for text in files”
 - Outputs lines which match the requested text
 - This can result in a lot of data...

```
boxcat$ grep penguin south.txt  
<MASSIVE WALL OF TEXT>
```


Useful Switches

- grep - “Look for text in files”
 - -i : Ignore case
 - -n : Print line number
 - -v : Exclude lines that match (invert)
 - -H : Always print filename (useful for multiple files)

Regular Expressions

- Want to search for more than just simple text sequence
- grep supports Regular Expressions
 - A way of expressing a search pattern
 - In fact grep means “Global Regular Expression & Print”

Regular Expressions

- grep uses special “metacharacters”
- . : Match any character
- ^ : Match only at the start
- \$: Match only at the end
- ... others (see man page)

Unix Pipelines

- Let's find the blank lines...

```
boxcat$ grep ^$ south.txt
```

```
<MASSIVE WALL OF BLANK LINES>
```


Regular Expressions

- Let's find the penguins twice..
- Once without regex & once with...

```
boxcat$ grep pengu south.txt
```

```
<MASSIVE WALL OF TEXT>
```

```
boxcat$ grep pen.u south.txt
```

```
<MASSIVE WALL OF TEXT>
```

- But what if we want to count them instead?

Unix Pipelines

- Recall the ideas
 - “Text streams as interface”
 - “Everything is a file”
- Idea:
 - Design programs to work either on files
 - Or a stream of lines of text
 - Feed the output of one program into the next

Unix Pipelines

- Revisit a grep example
 - Use the | to string together programs
 - Each program is a “filter”

```
boxcat$ ls -la | grep txt
-rw-r--r--      1 boxcat  staff          24 12 Jun 18:25 .sneaky.txt
-rw-r--r--      1 boxcat  staff    843763 12 Jun 18:24 south.txt
```

- Print out the ls entries for all the text files...

Unix Pipelines

- Let's count the penguins...

```
boxcat$ grep penguin south.txt | wc -l  
100
```

```
boxcat$ grep penguins south.txt | wc -l  
62
```


Unix Pipelines

- Combine with regexs.
- Let's count the penguins again ...

```
boxcat$ grep pen.u south.txt | wc -l  
102
```

Be Careful

- Did we really do it right?
- Let's check...

```
boxcat$ grep pen.u south.txt | grep -v pengu  
exploration, and all the parties will open up vast stretches of unknown  
forced the floes open until the 'Endurance' swung right round and drove
```

- Always make sure regex match what you think they do

Tools for the Pipeline

- `sort` - “Sort lines in file”
- `uniq` - “Remove duplicate lines from file”

Sort

- `sort` - “Sort lines in files”
 - Outputs the file lines in sorted order

Uniq

- Uniq - “Make lines unique”
 - Removes duplicate lines
 - Only: If they are adjacent
 - Combine with sort
- Useful switch
 - -c : Count

Lab - The Unix Environment

1. Get into groups

Introductory Python Programming

- The Python REPL
- Values and Variables
 - Numbers
 - Strings
- Decisions
- Text Editors
- Python Functions
- Other useful data types

What is Python?

- Python is a programming language
- Emphasises readability
- Easy to learn
- Imperative, OO & Functional Features

<http://learnpythonthehardway.org/book/>

What is Python?

- Python has 2 different versions
 - We'll use Python 2 (version 2.7)
- Several different implementations
 - We'll use the reference impl - CPython
 - Also Jython, IronPython, PyPy

<http://learnpythonthehardway.org/book/>

The Python REPL

- Read-Evaluate-Print-Loop
- Enter a line of code, instantly see the result
- REPL works one single input at a time
 - Saves state - e.g. variables
 - Contains definitions - e.g. functions
 - The underscore represents “last result”

The Python REPL

```
$ python
Python 2.7.5 (default, Mar  9 2014, 22:15:05)
[GCC 4.2.1 Compatible Apple LLVM 5.0 (clang-500.0.68)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print "hello world!"
hello world!
>>>
```

The Simplest Thing

```
$ python
Python 2.7.5 (default, Mar  9 2014, 22:15:05)
[GCC 4.2.1 Compatible Apple LLVM 5.0 (clang-500.0.68)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print "hello world!"
hello world!
>>> # Hello
...
>>>
```


Values

- A value is some data that we care about
- Python has many different types of value
 - Numbers
 - Strings
 - Lists
- Programmers can define their own types
- Let's meet some of the simplest ones...

Number types

- Integers
- Floating point
- Also:
 - Octal and Hex integer constants
 - Complex numbers

Integers

- Of arbitrary length
- No 32-bit (or 64-bit) limits
- Silently translates to long integer if needed
- 42 and -2^{128} are both valid

Floating Point

- Aka decimal numbers
 - 3.14159265359
- Floating point numbers are inexact
 - Rounding errors are inevitable
- “What Every Computer Scientist Should Know About Floating-Point Arithmetic”

http://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html

Arithmetic

```
>>> 3 * 4
```

```
12
```

```
>>> 3 + 2 - 5
```

```
0
```

```
>>> 7 / 3
```

```
2
```

```
>>> 7.0 / 3
```

```
2.3333333333333335
```

```
>>> 137 % 7
```

```
4
```

```
>>> 137 / 4
```

```
34
```

```
>>> 1E6
```

```
1000000.0
```

Bitwise Operations

$\ggg 2 \wedge 3$

1

$\ggg 2 \mid 3$

3

$\ggg 2 \& 3$

2

$\ggg 2 \ll 2$

8

$\ggg 16 \gg 2$

4

Strings

- Strings are encased within pairs of quotes
 - Can use either double or single quote marks
 - Can embed single quote mark of the opposite kind
 - `"Somebody 's"`
 - `'30" high'`

Strings

- Multi-line strings require triple quotes
 - `"""Somebody is creeping
quietly
up
the
stairs"""`
- Triple quotes can also enclose multiple quote marks
 - `"""Ed's Mum's friend said: 'It's cold'"""`

Special characters in strings

- The backslash character denotes an escape
 - Can be used to ignore a special character
 - `"Ed\'s Mum\'s friend said: \"It\'s cold\""`
 - Can be used to insert an escape code
 - `\n` is the escape code to insert a new line
 - `rhyme = "Jack be nimble\nJack be quick"`
`print rhyme` will return:
`Jack be nimble`
`Jack be quick`

Strings and concatenation

- Adjacent strings concatenate
 - `letters = 'a' 'b' 'c'`
`print letters` will return `abc`
- This is important to remember when passing strings to a function (see later)
 - `foo('a' 'b')` will pass the string `ab`
 - `foo('a', 'b')` will pass two strings, `a` and `b`

Variables

- Defined by *variable_name = value*
 - `start_value = 4`
 - `name = "fred"`
- Variable names
 - Can contain any combination of letters, numbers, or underscores _
 - Cannot start with a number

Variables

- Python is dynamically typed
 - This means values have types
 - But variables don't

```
>>> name = "Number Six"
>>> print name
Number Six
>>> name = 6
>>> print name
6
```


Decisions

- Model a decision
 - Aka a branching construct
- Need a condition
 - And consequences
- Loop Constructs

Decisions

- The if statement is used for simple decisions

```
>>> if age > 18 :  
...     print "Can drink in UK"
```

- The general form is:
- if <CONDITION> :

Python & Indentation

- if <CONDITION> :
 - The next line must be indented 4 spaces
 - You cannot safely mix tabs and spaces in Python
- More generally, each block must be indented correctly
- In REPL, have these prompts
 - >>>
 - ...

Decisions

- Can also have an else clause

```
>>> if age > 21 :  
...     print "Can drink in US"  
... else:  
...     print "Can't drink in US"  
... 
```


Elif

- If we have multiple if statements together

- Can use elif

```
>>> if age < 18 :  
...     print "Can't drink in UK"  
... elif age < 21 :  
...     print "Can drink in UK, but not US"  
... else :  
...     print "Can drink in both"
```

- What happens if multiple elifs are true?

Text Editors

- An effective way to work is to have a text editor
 - As well as a REPL
- Mac: Use any editor you're comfortable with
- Linux: Use any editor you're comfortable with
- Windows: Use gedit from within the VirtualBox

for

- for introduces a new kind of block
- for is used for a finite range

```
>>> for x in range(1, 4):  
...     print "We got %d" % (x)  
...  
We're on time 1  
We're on time 2  
We're on time 3
```

while

- while is similar to for
 - But when the loop is indeterminate
 - “Repeat until condition is met”
 - Equivalent to for in some circumstances

while

```
>>> x = 1;
>>> while x < 4 :
...     print "We're on time %d" % (x)
...     x = x + 1
...
We're on time 1
We're on time 2
We're on time 3
```

Python Functions

- Python allows us to create functions
 - Reusable code
- Use the def keyword
- `def <NAME>(<PARAMETERS>):`

Python Functions

```
>>> def drink(age):  
...     if age < 18 :  
...         print "Can't drink in UK"  
...     elif age < 21 :  
...         print "Can drink in UK, but not US"  
...     elif age > 65 :  
...         print "Can drink, and have a pension"  
...     else :  
...         print "Can drink in both"  
...  
>>>
```

Python Functions

```
>>> drink(37)
Can drink in both
>>> drink(77)
Can drink, and have a pension
>>> drink(17)
Can't drink in UK
```


Exercise 3

- Basic Programming Task
 - FizzBuzz

Other Useful Data Types

- Operating on Types
- Strings Revisited
- List
- Tuple
- Map

Operations on Data Types

- Python's data types support operations
 - Including String
 - Called “methods”
- Just append a variable with the operation
 - `<VARIABLE>.<OPERATION>`
- Let's see some examples..

Split

- Split apart a string on a substring

```
>>> x = 'blue,red,green'  
>>> x.split(",")  
['blue', 'red', 'green']
```

- Uses lists (see next slide)

Lists

- Lists are sequences of items
 - Lists are ordered
 - Lists can be mixed
 - Lists are mutable

Lists

- Defined by *list_name = [value, value, value, ... value]*
- `test_scores = [12, 19, 24, 15, 21]`
- `ingredients = ['eggs', 'butter', 'sugar']`
- Lists can contain numbers, strings, lists (or a mixture)
- `my_lists = [test_scores, ingredients]`
`print(my_lists)` will return
`[[12, 19, 24, 15, 21], ['eggs', 'butter', 'sugar']]`

List index positioning

- *list_name[x,y,z]* can be used to return multiple items
 - `pets = ['dog', 'cat', 'mouse', 'fish', 'bird']`
`print(pets[0, 3])` will return `['dog', 'fish']`
- *list_name[x:y]* can be used to return a range of items
 - `pets = ['dog', 'cat', 'mouse', 'fish', 'bird']`
`print(pets[1:3])` will return `['cat', 'mouse', 'fish']`

List index positioning

- `list_name[x]` returns a specific item from a list
 - `pets = ['dog', 'cat', 'mouse', 'fish', 'bird']`
`print(pets[2])` will return `mouse`
- `x` refers to the index position of the item
- In Python, list indexes **always start from 0**

List index positioning

- `list_name[x][z]` can be used to return an item from a nested list (i.e. a lists of lists)
- `x` refers to the position of the sublist within the main list
- `y` refers to the position of the item within the sublist
- ```
days = ['Mon', 'Tues', 'Wed', 'Thur', 'Fri']
months = ['Jan', 'Feb', 'Mar', 'Apr', 'May']
date_lists = [days, months]
print(date_lists[1][4])
```

 will return **Apr**

# List operations

- `list_name.append(item)` adds a single item to a list
- ```
ingredients = ['eggs', 'butter', 'sugar']  
ingredients.append('flour')  
print(ingredients)
```

 will return:
`['eggs', 'butter', 'sugar', 'flour']`
- When using `.append` new item is added at end of list

List operations

- `list_name.extend(other_list)` can be used to add the items from one list to another list
- ```
ingredients = ['eggs', 'butter', 'sugar', 'flour']
toppings = ['icing', 'cherries']
ingredients.extend('toppings')
print(ingredients)
```

 will return:  

```
['eggs', 'butter', 'sugar', 'flour', 'icing', 'cherries']
```
- New items appear at the end of list
- In the same order as the original list

# List arithmetic

- Arithmetic functions can be also used to add the items from one list to another list
- `ingredients = ['eggs', 'butter', 'sugar', 'flour']`  
`toppings = ['icing', 'cherries']`  
`print(ingredients + toppings)` will return:  
`['eggs', 'butter', 'sugar', 'flour', 'icing', 'cherries']`
- `ingredients = ['eggs', 'butter', 'sugar', 'flour']`  
`toppings = ['icing', 'cherries']`  
`cake_mix = [ingredients + toppings]`  
`print(cake_mix)` will return:  
`['eggs', 'butter', 'sugar', 'flour', 'icing', 'cherries']`



# List operations using indexes

- `del list_name[x]` removes an item from a list
  - `pets = ['dog', 'cat', 'mouse', 'fish', 'bird']`  
`del pets[2]`  
`print(pets)` will return `['dog', 'cat', 'fish', 'bird']`
- `new_item list_name[x]` replaces an item within a list
  - `pets = ['dog', 'cat', 'fish', 'bird']`  
`pets[3] = 'hamster'`  
`print(pets)` will return `['dog', 'cat', 'fish', 'hamster']`

# List operations using indexes

- `list_name.insert[new_item, x]` inserts an item into an indexed position in a list
- ```
pets = ['dog', 'cat', 'bird']  
pets.insert('monkey', 2)  
print(pets) will return ['dog', 'cat', 'monkey', 'bird']
```
- `list_name.reverse` reverses the order of a list
- ```
pets = ['dog', 'cat', 'bird', 'snake']
pets.reverse
print(pets) will return ['snake', 'bird', 'cat', 'dog']
```



## More about nested lists

- Example: creating a list containing numbers, strings and sublists with a single command
- `my_list = ['raindrops', 7, 'kittens', ['girls', 'dresses', 'white'], 42, 'sleighbells']`  
`print(my_list[1:3])` will return  
`[7, 'kittens', ['girls', 'dresses', 'white']]`
- `print(my_list[3][1])` will return  
`dresses`

# Tuples

- Defined by *tuple\_name* = (*value*, *value*, *value*, ... *value*)
  - `reference = (126.629, 193.620, 242.117)`
  - `elements = ('earth', 'air', 'fire', 'water')`
- What is the difference between a list and a tuple?
  - Tuples are immutable – once created, they cannot be changed
  - Tuples are faster than lists
  - Tuples can be used as map indexes (we'll come to maps next)



# Maps

- Maps (or dictionaries) are like look-up tables
- Think of them as sets of pairs
  - Each pair consists of a key and a value
  - You use the key to retrieve the value
  - Keys can be strings, integers or tuples – they must be immutable (fixed)
- Maps are unordered

# Maps

- Defined by *map\_name* = {*key* : *value*, *key* : *value*, ... }
- `fav_colour = { 'jack' : 'red', 'sam' : 'blue' }`
- `my_street = { 12 : 'Smiths', 18 : 'Jones', 22 : 'Evans' }`
- Can also be defined using the dict construction
  - `dict ( [ ('apples', 4), ('pears', 3), ('lemons', 8)] )`
  - `dict = neighbour (name:'bob', age:43, kids:('tom', 'jo'))`



# Map operations

- `map_name[x]` returns the value mapped to the key 'x'
- `fav_colour = { 'jack' : 'red', 'sam' : 'blue' }`  
`print(fav_colour['jack'])` will return **red**
- This operation can be used with nested maps
  - `fruit = { 'pears' : 3, 'apples' : { 'red' : 3, 'green' : 1} }`  
`print(fruit['apples']['green'])` will return **1**

# Map operations

- `map_name.keys` returns a list of keys within a given map
  - `scores = {'jack': 13, 'sam' : 8, 'beth' : 17}`  
`print(scores.keys)` will return `['jack', 'sam', 'beth']`
- `map_name.values` returns a list of values within a given map
  - `prices = {'coffee': 4, 'sandwich' : 7, 'steak' : 15}`  
`print(prices.values)` will return `[4, 7, 15]`



# Map operations

- Maps use many similar operations to lists
- A map operation uses a key as a reference
- `del.map_name[key]` deletes a specific key/value pair
  - `scores = { 'jack' : 13, 'sam' : 8, 'beth' : 17 }`  
`del.scores['sam']`  
`print(scores)` will return `['jack' : 13, 'beth' : 17]`
- `map_name[key] = [new_item]` replaces the value assigned to a key
- `map_name[new_key] = [new_item]` adds a new key/value pair

## Exercise 3

- Basic Programming Tasks
  - Generating Fibonacci Numbers
  - (Bonus) Largest Palindrome Product



# Python Programs as Linux Scripts

- How Python handles STDIN, STDOUT, STDERR
- Python Regular Expressions
- Python File Handling
- Handling CSV Files
- Example (iTunes)

# Python Standard Filehandles

- `print` writes to `STDOUT`
- `input` (& `raw_input`) read from `STDIN`
- How do we write a Python “filter”?
  - Use `sys.stdin` & `sys.stdout`



# Python Regular Expressions

- Python has extensive & powerful regex capabilities
  - Extend the capabilities of grep
- Uses a module (see tomorrow)
- More power

# Python Regular Expressions

- Do more than the grep syntax
- A consensus of “extended regex syntax”
  - Started by Perl
  - Other languages adopted the conventions



# Python Regular Expressions

- Support all the grep capabilities
- `+` : 1 or more
- `?` : 0 or 1
- `{m}` : Exactly m copies
- `{m,n}` : Between m & n

# Python File Handling

- Simplest form
- Open for reading

```
f = open(file_name)
for line in f:
 print line;
f.close
```



# File Writing

```
f = open("example.txt", "wb")
print "File name: ", f.name
```

```
Close file
f.close()
```

# Handling CSV Files

- Simple file format
  - Separated by commas
- William,Shakespeare,1564,1616



# Handling CSV Files

- Simple file format
- Separated by commas
  - Sounds simple
  - But what if data contains commas?
    - How do we cope with this?
    - Pipelines?

## Exercise 4 (ITunes)

- tracklist.csv
- What can you find out from it?
  - E.g What bands are most listened to?
- Can you read into a map?
  - Or a list of maps?



# THANK YOU

**Commercial** - [www.jclarity.com](http://www.jclarity.com) @jclarity

## Products

jClarity Censum: The world's best GC log analysis tool

jClarity Illuminate: The learning performance problem finder

**Community** - [www.meetup.com/londonjavacommunity](http://www.meetup.com/londonjavacommunity)

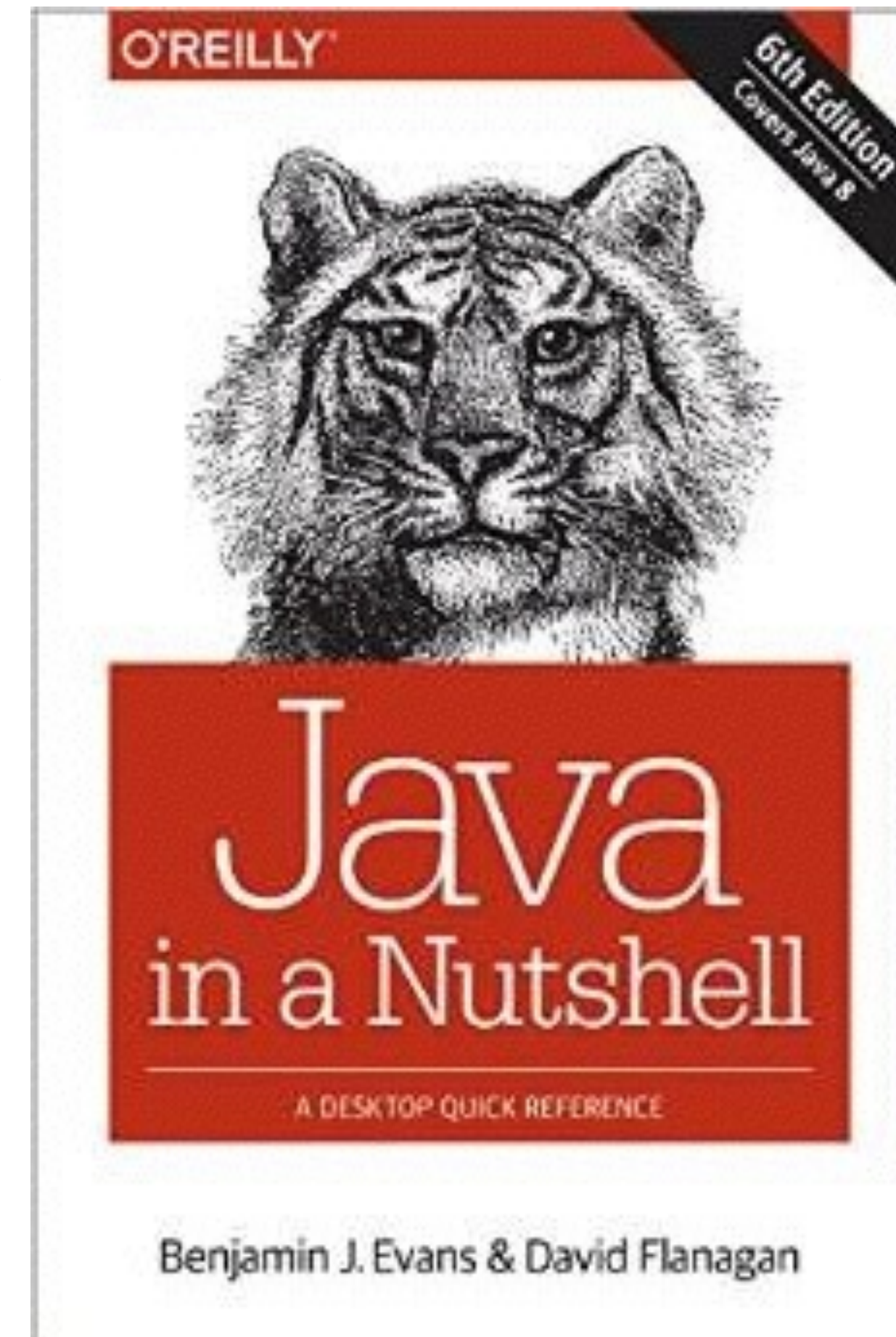
**Email** - [ben@jclarity.com](mailto:ben@jclarity.com)

## Books:

Java in a Nutshell (6th Edition) - O'Reilly

The Well-Grounded Java Developer - Manning

Optimizing Java Performance (Forthcoming)





# Acknowledgements

"Unix history-simple" by Eraserhead1, Infinity0, Sav\_vas - Levenez  
Unix History Diagram, Information on the history of IBM's AIX on  
ibm.com. Licensed under CC BY-SA 3.0 via Wikimedia Commons -  
[http://commons.wikimedia.org/wiki/File:Unix\\_history-simple.svg#/media/File:Unix\\_history-simple.svg](http://commons.wikimedia.org/wiki/File:Unix_history-simple.svg#/media/File:Unix_history-simple.svg)