Programming and programming languages

vishvAs vAsuki

June 3, 2013

Contents

Part I

Introduction

1 Research themes

1.1 Programming language design

How would you program a program to solve some problem in a very simple way? Make it as simple as possible for the programmer to specify his intention without ambiguity; then translate to the computer.

2 Language Design concepts

Programming is data processing.

2.1 Chapter Scope

Programming language features enabling different programming paradigms and architectures are considered here, while the the higher level programming paradigms themselves are considered elsewhere. Eg: Higher order functions underlies functional programming paradigm.
For details about threads etc.., see software systems ref.

2.2 Syntax

2.2.1 Literals

All literals, apart from comments, in a program are of the following sort: keywords; identifiers (names of variables); operators. Literals are separated using delimiters and by using rules which specify the allowed structure of the literal.

2.2.2 Identifiers

2.2.2.1 Overloading

A language may allow methods/ operators with similar names, but different signature.

2.2.2.2 Scope

Namespace/ scope of variables and functions is ordinarily defined by the block of code wherein it is defined.

2.2.2.3 Implicit scope

But, in some cases, a variable may be defined to have 'implict' scope. Such variables are automatically used by the compiler when necessary - example to bind an argument not explicitly passed to a function, to convert between data-types.

2.2.3 Expression evaluation

2.2.3.1 Lazy vs eager evaluation

In lazy eval: length([2+1, 3*2, 1/0, 5-4]) returns 4 without even bothering to evaluate the values in the array. This difference in evaluation becomes important when the computations involved are costly: eg: function calls.

2.2.4 Syntactic sugar

Some languages, for the convenience of the programmer, define shorter ways of writing various frequently used expression types.

2.2.4.1 Macros

Sometimes, a user may add to syntactic sugar by the definition of 'macros' or expressions which are evaluated before the program is even parsed by the compiler.

2.3 Abstraction from hardware

2.3.1 View of memory

If in writing code, one thinks in terms of instructions and data which can be understood by a processor (eg: bytes in the main memory, registers, hardware IO signals), one is working with a low level language.
High level languages allow the programmer to think of memory locations and data more abstractly/ declaratively by starting with assumptions that there exist memory locations with certain properties which will be bound to variables used by the programmer by the compiler with the OS's help.
Mixed level languages allow a combination of both these paradigms.

2.3.2 Portability

Programs written in high level language are more portable across computers with different architectures and abilities due to the existence of compilers or interpreters or virtual machines implemented on various platforms which suitably translate a high level program to something understandable to the processor.

2.3.3 Programming discipline

High level languages often try to enable/ enforce some programming discipline (eg: by enforcing OOP paradigm, functional programming, variable scoping etc..), provide more concise commands for frequent operations, perhaps using 'syntactic sugar'; though internally they may be forced to use programming paradigms different from the programming interface provided to the programmer.

2.3.4 Type system

2.3.4.1 Implementing abstract data-types

While programming, rather than think of data in terms of bytes, one can consider and manipulate data (collection of data) at a more abstract level. One can define different generic operations and views on a set of bytes depending on the type of data stored therein.
Examples of data-types may be Integers, Floating point numbers or even arbitrary user-defined classes. A very basic datatype offered by high-level languages is the Array/ sequential storage.

2.3.4.2 Compile time vs run-time

Aka Static vs dynamic.
In static type checking, data types are associated with variables, and these are checked at compile time to ensure that assignments do not violate this binding. So, the binding of a type to a variable happens at compile time, rather than run time. This enables us to catch errors and prove more properties about a program's correctness before it is compiled and run. It also enables safer refactoring of code (changing type to a subtype etc..).
In case of dynamic typing, types are associated with values rather than variables. THis allows the programmer more flexibility in his use of variables.

2.3.4.3 Strong vs weak

Can the type of data pointed to by a variable symbol change within its scope? Scala has weak typing and static typing. Python has weak typing and dynamic typing.

2.4 Procedures

Code is often organized into different procedures, which may be invoked under different contexts with different arguments.

2.4.1 Mathematical functions

A procedure can change state (has a side-effect) and maybe return a value, doesn't naturally correspond to mathematical function which cannot do this. So a function, which cannot have side-effects, can be viewed as a special type of procedure.
C is a procedural programming language which is not structured.

2.4.2 Passing arguments to procedures

Variables can be passed by reference, or by value. In the former case, an address to the data can be considered as being passed; so a modification to the data will be visible after the function returns. In the latter case, this is not true.

2.4.3 Operators

Operators are special functions, often with special syntax. They sometimes implement logic which is so finely grained that it is executed in a lower level programming/ machine language.

2.5 Object creation and manipulation

Object oriented design is considered in the software architecture survey.

2.5.1 Object manipulation

The main idea of object oriented programming is to describe computation in terms of manipulating states of objects. This causes state changes to be local - which helps programmers design a well structured view of state and behavior. This is distinct from the functional programming paradigm.

2.5.2 Encapsulation and locality

Hiding data and methods from outside methods restricts state changes to be local.
For enforcing varying degrees of locality, Method/ property access specifiers may be provided: public, private, protected, default.

2.5.2.1 Methods

Methods: procedures associated with a class. The behavior of a method within a certain object obj - say obj.f(argList), in terms of functional programming, can be considered to be a function which accepts two argument sets: first is the object obj, and the second is the argList passed to the method; and which then returns values which include the modified object, which is then assigned to the appropriate reference.
Maybe class has events, listeners too.

2.5.3 Object construction and destruction

There are various patterns for specifying functions which create objects based on prototypes.

2.5.3.1 Classes

Based on their state-space and purpose, these objects may be described as instances of various classes, in whose definition various attributes and methods may be defined.
Use of classes enables static type checking.

2.5.3.2 Cloning prototypes

One way to create objects is by creating a clone. This mechanism works even where objects' types are not well defined.
Properties and methods are then added as desired.

2.5.3.3 Deep copy

With a deep copy, not only is a new object with the same state as the original object created, but the following happens too: any object contained within the original object is also deep copied (rather than both objects referring to the same member object).

2.5.3.4 Factory methods

Often easily accessed factory methods are provided in order to create an object of a certain class with a certain state. When these are member methods of the class in question, they are called constructors.
When an inheritance structure is present, and when a parent class has multiple constructors, it becomes necessary to ensure the calling the right parent constructor. Sequence of calling constructors: superclass first. Destructors: subclass destructor called first.

2.5.4 Inheritance

Among classes of objects, a inheritance structure may be defined - Eg: class A extends class B in that it has all the members in B (with identical or overriden semantics/ behavior), together with some extra members.

2.5.4.1 Benefits

Inheritience is compared with object composition elsewhere.
This accomplishes two things: 1] It is a neat code-reuse pattern; and 2] enabling static type-checking (in case of statically typed languages) - ie when a member x is accessed, ensuring that the class has an appropriate member x.

The fact that members of objects of different subclasses of the same superclass can be accessed using a variable (of the superclass type) is called polymorphism.

2.5.4.2 Abstract class/ interface

Programming laguages often allow definition of types which cannot be instantiated, but can be used to define the members of other classes which extend/ implement them. So, these abstract classes need not define a generic implmentation of methods they declare.
This enables one to define functions which accept arguments with a certain 'interface'.

2.5.4.3 Dynamic dispatch

When a superclass and a subclass have a function of the same name f, it is generally intended that subclass.f should be called when obj.f is invoked, where obj is declared to be of the superclass type.
Some languages require this to specified explicitly using the keyword virtual.

2.5.4.4 Multiple inheritance

In case multiple ancestors (along distinct inheritance paths) of a class C define a method with the same name m, the effect of the call $C.m$ needs to be disambiguated explicitly by the programmer: and the programming language should ideally provide him the ability to do so.
In case of programming languages which do not allow this, one must instead use interfaces with object composition as a way of sharing code.

2.5.5 Prototyping-based languages

These do not provide a mechanism to explicitly define a class or inheritance structure. Objects are created exclusively by cloning and dynamically (at runtime) adding members - Classes/ inheritance are implicitly defined in this process.

2.5.6 Mutability

Some objects may be designed to be immutable (having unchangeable state). Such objects may have 'state-altering' methods which construct and return an object with the altered state.
So, object-cloning is superfluous for such objects.
Example: Immutable maps, when return a copy of value objects to ensure that the value is not modified.

2.5.7 Class with optional members

Aka algebraic data-type. An algebraic data-type a class which is mostly defined by the members it does and does not have. These members are often set using constructors.

2.5.7.1 As inheritance tree

Often it is defined by an inheritence hierarchy.
eg: trait Tree; class BTree(lTree: Tree, value: Int, rTree: Tree) extends Tree; class LeafNode(value: Int) extends Tree.

2.5.7.2 Use with pattern matching

An instant of an algebraic data type is often matched using a special pattern matching construct provided by the language.
eg pseudo-scala-code:

```
f(tree) = tree match {
case BTree: print(tree.lTree) f(tree.lTree) f(tree.rTree)
case LeafNode: print(tree.value)
}
```

2.6 Functions as objects

Functions can be used as arguments and return values. So, functions can be regarded as objects.
Several languages provide excellent support for easily defining such functions.

2.6.0.3 Functions acting on functions

Aka Higher order functions. Easy to define higher order functions: Eg: $\frac{d_n}{dx^n}$.
So, one can conveniently pass a function as an argument to another function.

2.6.0.4 Event driven programming

One can view some computation in terms of events and event listeners/ handlers/ callback functions which might handle them. A 'main loop' keeps polling for events and dispatching events to the correct handler. This can be easily realized with functional programming.

2.6.1 Functions with non-local references

Aka closures.
With closures, a function definition may include references to non-local variables. The function object is created/ updated anew just before it is called, so that changes to the non-local variable alters the function definition.

Some languages even allow the function definition to update the value of these variables.

2.6.1.1 Extended function definition using decorator

Consider a higher order function h() accepting and returning a function. The statements f = someFunction and f = h(f) is given syntactic sugar using decorators: see python chapter for example.

2.6.2 Partial application

From the definition of a function f over k arguments, one can derive definitions of other functions depending on $l < k$ arguments which arise as a result of fixing a value to one of the arguments of f. Enabling this with simple syntax is an attractive feature in many languages.

2.6.3 Currying

Any such vector-domain function can be described as a function f' acting on a scalar domain (corresponding to the first dimension of the domain of f) and producing a
vector-valued function: $f' : D \to G$, where $G = \left\{D^{k_{in}-1} \to D^{k_{out}}\right\}$ and $f'(a) = g \in G$ such that g corresponds to the partially applied function $f(x_1 = a, x_2..)$.
Doing this recursively, the entire computation can be written in terms of scalar functions!

2.6.3.1 Lambda calculus

Lambda calculus provides a good representation and manipulation rules for considering computation in this way. It describes computation in terms of formulae (unnamed functions) involving many variables, which may be composed with each other; whose free variables may be bound to a certain value etc..

2.6.4 Iteration aids

Functional programming, in its purest form, does iteration using recursions.
Plus, various higher-order functions which can be defined to act on lists and iterators enable succinct definition of recursive computation.

2.6.4.1 Tail recursion

A tail recursion is a recursive function where the last statement executed before the function exits (which is not necessarily the same as the last statement in the function definition) involves a simple recursive call. The value returned, if any, is simply the unmodified result of this recursive call.

Reuse of stack space Ordinarily an additional recursive call would involve addition of new memory in the call-stack space. This can be avoided in this case by simply reusing the stack-space already allocated to the calling function. When the language standard does not demand it, but is implemented by a certain compiler, this deserves to be called an 'optimization'; otherwise it is an integral feature of the language.

Advantage This allows iteration to be written using a recursive call without the penalty of using space linear in number of iterations.

2.7 Functions without side effects

Computation is the evaluation of math functions.
State changes outside the function, other than sending a return value is not allowed: functions without side effects.

2.7.1 Dependence on non-input values

To avoid side effects, some languages may force function behavior depends purely on input: there is no access to higher-scope variables (eg: global variables in C).

2.7.1.1 Closure restrictions

However, other languages may allow function-behavior to depend on non-input values, but may not allow altering them within the function. Eg: Matlab, R.
Such non-input and non-local variables and values in the function definition are called free variables (- similar to the definition in case of first order logic).
These free variables may be bound (assigned values) at run-time, just before the function is to be invoked.
Thus the precise definition of a function is finalized only at run-time - So functions with free variables essentially define a function family with a method for picking the right function from this family given some values.

2.7.2 Immutability of input

'Pass by reference' is not allowed as that would allow code in the function to have side-effects.

2.8 Decision structures

The conciseness and clarity of expression of branching and iterative operation in different languages is different.

2.8.1 Branching

The if-else structure is the most basic decision structure, using which more convenient decision structures like the if-elseif.. structure is defined.

2.8.1.1 Matching a value

Different branches of execution often depend on different tests performed over the value of an expression. This is provided in terms of the 'match with various cases' construct.

2.8.2 Iterative operations

The essential ingredients of an iteration are the starting state, the code to be executed and the termination condition.

2.8.2.1 Imperative loops

A basic imperative programming construct for iteration is the while loop. To handle various specialized cases more concisely, constructs such as C's do-while and for loops are defined.

2.8.2.2 Recursion

The most basic functional programming construct for iteration is recursion, with efficient reuse of the stack space using tail recursion where possible. Concise definition of recursive functions is possible with the use of branching based on matching values- as in Scala.

2.8.2.3 Declarative operations on lists

One can then define higher-order functions which operate on iterators and lists leading to very concise statements. These are described elsewhere.

2.9 Type-generic definitions

One may define generic classes or functions in a generic way. Such definitions accept a type parameter, apart from other values. Eg: class List[T] may define a linked list which holds objects of type T.

2.10 Specifying generators

These are functions which produce the ith element only on demand - a sort of lazy evaluation. They lead to time savings.
These are often traversable only once. Some languages allow a succinct notation for producing a list by executing some code for various elements drawn from an iterator.

2.10.0.4 Set-builder syntax

Aka comprehension. This is not equivalent to a for loop in C, but a consise way of specifying a list literal.
Eg: The for comprehension in scala.

2.11 Application programming interface (API)

To aid rapid software development, many programming languages come with libraries of objects and functions implementing common data structures like collections, file parsing etc..

2.11.1 Higher order functions for lists

Having an API with support for higher order functions acting on iterators and lists enable writing concise code.

2.11.1.1 Map

map(iter, fn) applies the fn to each element produced by an iterator and returns a iterator of results.

2.11.1.2 Filter

This produces an iterator with elements of another sequence which pass a test specified by a certain function.

2.11.1.3 Transformation-types

In case of functions which produce new lists/ iterators such as maps, fitlers, one might either want the transformations to take effect one at a time while an element is being retrieved (lazy/ non-strict), or one might want a function which produces the result list/ iterator fully upon call. The former is more efficient (especially when avoiding intermediate results when doing x.map().fitler..), while the latter may be desirable to preserve modularity.

2.11.1.4 Fold

foldLeft(list, fn, b): Let head(list) be the first element in the list and tail(list) be the list without its first element. Then, if list is empty, foldLeft returns b; otherwise, foldLeft(tail(list), fn, fn(b, head(list))) is returned.
foldRight: is a function which does the same, but from the other end of the list: so one needs to modify definitions of head and tail appropriately in the above description.

2.12 Remote procedure call (RPC)

Allows a process (RPC client) to run a procedure within a remote process (RPC server) is such a way that the programmer does not have to worry about details of how the arguments/ results are communicated etc.. - from the perspective of the programmer, this call should be as simple as a local procedure call. In case of synchronous calls, the calling process blocks until the remote procedure terminates and returns the result.

2.12.1 Pipeline

Client code - RPC client stub - RPC server stub - Server procedure.
Client code - client stub and server code - server stub calls are done using normal (local) procedure invocation mechanisms. The purpose of client stub and server stub is to communicate the arguments/ results returned to/ from them by their respective processes - this involves marhsalling locally received objects and unmarshalling messages from the other stub into objects. Client stub - server stub communication is done through an internal protocol which the calling function need not know.

2.13 Exception handling

Here we only consider exception handling features specially built into the language. For other constructs, like monads and error codes, see the software architecture survey.

2.13.1 try catch blocks

This is exemplified by the try - catch - finally block of Java.

2.13.1.1 Context handler classes

[**Incomplete**]

2.14 Test framework

Desiderata for tests are listed in the software architecture survey.
Various test-cases may use a single object. So, to achieve speed, they may share a test fixture, which contains the shared object - rather than initialize it multiple times.

3 Parallel programming programming models

General parallel programming concerns are considered in the distributed computing survey.

3.1 Thread abstraction

Parallel programs can be considered in terms of threads of computation. When data is shared, one should take care of race conditions.

3.1.1 Fork and join

With this paradigm, a thread can fork to form a new thread. [**Incomplete**]

3.2 Mapreduce

3.2.1 Computation flow

3.2.1.1 Input preparation

Master splits the input into multiple chunks, as necessary. It collects counter updates from mappers and reducers. May provide a html status page.
If applicable, the master also joins disparate input sources to produce a (key, valueList) pair.
There can be multiple disparate input source tables - each may be fed to a different mapper. Outputs from disparate mappers are combined based on key before reduce stage.

3.2.1.2 General flow per mapper

(key, value) - mapper - ((key, value)), counter updates - [combiner - ([same key], valueList)]
A combiner is a special type of reducer which may combine values from the outputs of a single mapper task before sending them accross the network for combination by reducers.

3.2.1.3 Flow per reducer

There is one shuffler per reducer. One reducer per shard.
(key, value) from various mappers - shuffler - (key, valueList) - reducer - ([same key], valueList), counter updates in output (file) shard.

3.2.1.4 Output storage

A file location in a distributed file system, together with the desired number of shards and output format is usually specified.

3.2.1.5 Thread safety

There is one thread running through the map function generally. So, in case of object oriented programming languages: the mapper object is usually thread-safe, Although class variables are not.

[**Incomplete**]

4 Translation to machine code

4.1 Machine instructions and their files

Machine instructions are understood by the processor. Assembly level code provides a way to write machine instructions using words instead of hexadecimal instructions.

4.1.1 Executable files

When an operating system is being used, machine instructions, which are stored in a file or on fixed locations in the hard-disk, should be associated with suitable meta-data/ headers. Such files are called executable files.

4.1.2 Object files

Object files contain named segments of machine instructions - library functions. These files may be linked to executable files, from which, using a mechanism like the 'call stack', data is processed using the library functions.

4.2 Translation from High level programming language

Writing machine instructions directly is excruciating - most of the work is mechanical - it is best done automatically by specialized programs which read files with precise yet high-level instructions and create machine instructions.

4.2.1 Compiled vs interpreted languages

In the case of interpreted languages, this is done one line at a time. In case of compiled languages, this is done one program or code block at a time.
Some languages, like Java are compiled to a byte-code, which is then interpreted (translated to machine language) during execution.
Run-times of programs written in Interpreted languages can slow down due to the cost of translating each line of code at run-time. This problem is assuaged with the use of 'just in time' compilers - now, loops can potentially be translated entirely to machine language before they are entered.

4.2.1.1 Freedom from error

Compiled langauges tend to be more error free, because successful compilation guarantees: a] every variable used is known to be clearly defined. b] Every function is passed the right sequence of arguments.

In case of a interpreted language, since they lack the compilation checks, errors are caught only at run-time, that too if execution of the erroneous part of the code is attempted. This can be assuaged by use of special linting tools.

4.2.2 Steps in translation

During lexical analysis, lexical tokens are identified from the high level program; variable names are distinguished from constants and operators. During syntactic analysis, or parsing, the tokens are put together into a bunch of instructions. Finally, addresses are assigned to variable names, and translation happens.
During this, linking is done by a linker: references to functions described in other object files are resolved.

4.3 Memory allocation

4.3.1 Compile-time memory allocation

This is memory allocated to data at compile time.
Memory can be considered as having been allocated at the time of function call and deallocated when the funciton exits. So, this is often visualized/ reffered to/ implemented as 'stack space'.
Stack memory is often more limited compared to heap space (although both can be increased).

4.3.2 Run-time memory allocation

This is done at run-time. One cannot be absolutely certain beforehand about the exact amount of memory a program will require. There are overheads in dynamic allocation: there can be problems such as fragmentation: data used by the same chunk of code may be located in different memory blocks (either in the main memory or in the cache on the hard disk). Allocation algorithms take atleast 50 instructions for making one allocation.
The space used for dynamic memory allocation is often called 'heap space'.

5 Character encoding

Displaying, accepting and writing to files characters readable by humans are common tasks in many programs, irrespective of language.
So, common standards have evolved to represent these characters with natural numbers or characters visible on most English keyboards or arbitrary bytes.

5.1 ASCII

This represents the Latin alphabet plus some common characters. Range: 0:255.

Special characters include control characters.

5.1.1 Control characters

Carriage return (move cursor to beginning of current line), line-feed (start a new line), tab (a long horizontal space). Their popular latin/ symbolic representations are: '
r
n
t'. Carriage return without line-feed is often used to overwrite text.

5.2 Unicode

This is capable of accomodating symbols in many of the world's scripts. It uses multiple bytes. Since the number of bytes actually needed may vary, variants such as UTF-8 are used.
UTF-8 is backward compatible with much of ASCII. If extra bytes are needed, it is indicated by a special bit.

Part II

Development tools

6 Build tool design

6.1 Build Targets

A programmer may want to do certain specific actions with his code.
For example, downloading dependencies for being able to compile, compiling code, running test cases, running the code, packaging the code, deploying a package on a web-server. These actions are called build targets.

6.1.1 Continuous mode

With some dependencies, one can execute a specific action while constantly scanning for source code changes..

6.1.2 Target Dependencies

There may be various dependencies amongst these build targets. For example, to compile one may need to download some libraries from the internet.

6.1.3 Library dependencies

Some build tools offer automated library dependency management, where if a repository and the dependencies are specified, the dependencies are automatically downloaded from the internet.

6.2 Quality metrics

A good build system provides a concise way of declaratively expressing actions to be performed for various build targets: So, various commands for actions like downloading should be available.
Some build systems are also integrated with IDE's'.

7 Unit test tools

7.1 Mox for Python

The tester stubs out various functions that the method being tested is supposed to call. Then in the test, you make calls to those stubbed out functions with the arguments that you expect to be passed to them if the function being tested is working properly, and mox records it. Then, when the function is actually called, mox asserts that the order and arguments to the stubbed out functions are what was expected.

8 Build tools

8.1 Cross-platform makefile generator

CMake uses makefiles, wherein compilation tasks are defined. It can be used to generate eclipse and visual studio projects - eg: with cmake -G"Eclipse CDT4 - Unix Makefiles" dirPath.
It is configured using cmakelist files.

8.2 sbt

Scala build tool is written in scala. Configuration files tend to be very concise. See a ready empty sbt project for a quick start.
Important commands include: update (To satisfy dependencies), actions, compile, package. Some commands may be prefixed by $\sim$ (Which constantly scans for source code changes and recompiles automatically when necessary.)

8.2.1 Dependencies - automation

One can automate the problem of downloading certain versions of external libraries on which the project depends, and adding the jars to the CLASSPATH if necessary.

8.2.1.1 Repository specification

By default, Maven2 and Scala Tools Releases repositories are used. The former includes many sourceforge libraries. One can add other repositories: resolvers += name at location.

8.2.1.2 Library specification

In build.sbt in the project folder, add the following line:

```
libraryDependencies += groupID % artifactID % revision
```

If you are using a dependency that was built with sbt, double the first % to be %%.
The revision argument could be: 1.3.2 or "[1.3,)" etc..
One can explicitly specify the url above by adding 'from "url1"' to the line above. The url is used only if it is necessary.
One can also add 'sbtAction' to indicate that the library dependency need only be included if a certain sbtAction is to be performed. [**Check**]

8.2.2 Tasks: definition

lazy val collectJars = task { collectJarsTask; None } dependsOn(compile)
Here, collectJarsTask is a scala method.

8.2.2.1 File IO tasks

Use FileUtilities.

8.2.2.2 Documentation

sbt doc

8.3 maven

Build configuration files are in xml, which tends to lead to bloated configuration files.

9 Task management

These services/ software are used to manage bugs, features, issues. Issue management is often bundled with project hosting websites which offer version control.

10 Version control software

10.1 svn

General syntax: svn command arguments.
For help, see svn help.
Commands include: commit/ ci, diff, update/ up, merge, resolved, status/ st.
To rollback a file to a previous revision: svn merge -r newRev:oldRev filename.
`export SVN_EDITOR=vi`

10.2 git

Enables distributed development by letting people maintain local repositories which may be merged with a central repository periodically. It also emphasizes security, speed. It also enables non-linear development by allowing branching and merging.

10.2.1 Architecture

There is a remote repository R, local repository L and a cache/ local index/ changeset C. Finally, there is the working directory W.
The repositories may contain various branches besides the master branch.

10.2.1.1 (Un)Ignoring changes

Add untracked files to .gitignore.
For tracked files:

```
git update-index --assume-unchanged file
git update-index --no-assume-unchanged file
git ls-files -v|grep ^[a-z]

git update-index --skip-worktree [FILE]
git update-index --no-skip-worktree [FILE]
git ls-files -v|grep ^S
```

10.2.1.2 Repository commits and logs

Every change committed to a particular repository is given a unique hash (we call this hashname) Eg: 53da224. One can use this as an identifier to rollback or compare changes.
The sequence of commits is stored in two places: logs and reflogs. Issuing 'rebase' commands, for example, fixes logs so that they reflect the desired

sequence of check-points; yet it does not touch the reflogs sequence of commits. Thus the older history is retained. The reflog is periodically cleaned up to concur with logs.

10.2.2 Update Commands

git branch branchName creates a new branch.
C can be updated using add (for both new and modified files) and rm commands. git -A adds all altered files not excluded by .gitignore.
C can be commited to L using commit. L is synced with R using push and fetch.
R is synced with (W, L) using pull. By default the master branch is used, but one can specify a branch as an argument.
W can be created from R using clone. W can be checked out from L using checkout.

10.2.2.1 Merges and stashes

Differences between local and remote branches show up when doing a pull. One cannot push changes to a repository without first pulling and merging locally. Conflicts between W and R may be resolved by the following action sequence: a] git stash b] git pull c] git stash apply - or in case the stash is disposable, git stash merge.
git pull –rebase pulls remote changes and does surgery on local git history to put your local changes on the top. This is dangerous - you can loose previous local commits. If that happens, one can use git reflog to retrieve the hashname for the change; and then do git checkout (if that hashname has not been garbage-collected).

10.2.2.2 Checking differences, tool configuration

git difftool [-t kdiff3 HEAD∼2]
git config –global merge.tool kdiff3 (creates and) modifies .gitconfig file in the home directory.

10.2.3 UI

git branches may be visualized using a UI.
gitk (optionally supplied a path-name) does this. gitk –all shows all branches. Thence git-gui may be started, which provides a good UI for updates and commiting.
git-gui provides a good ui for examining changes and checking in. One can add tools: for assume-unchanged $FILENAME, for example.
[**Incomplete**]

10.3 hg / mercurial

For help, see hg help. Similar to git.
General syntax: svn command arguments.
Commands: clone repo [destination]: checks out a repository to create a 'local' repository.
status/ st.
commit/ ci -m "msg": commits to the local repository. push: commits to the central repository.

10.4 Version control websites

- sourceforge uses svn.
- Google projects uses hg, svn and git.
  Shows activity level, number of active contributors, lines of code prominently.
- github hosts git projects
- bitbucket hosts hg projects.

11 Visualization toools

11.1 Graph

Gephi does the job, and supports a wide variety of input file formats including csv edge list.

12 Integrated Development Environment (IDE)

12.1 Eclipse

Eclipse allows development in many languages: C/ C++, Python, Java, Scala.

12.1.1 Startup configuration

This can be done in the eclipse.ini file. Heap size can be increased using -Xmx2048m after -vmargs. [**Check**]Is it necessary to set heap size in the run configuration too?
Java library path can be specified by -Djava.library.path=path.

12.1.2 Installing and updating components

Various plugins and the core software itself can be installed and upgraded using the GUI, after first enabling various online repositories using another UI.

12.1.3 Views and perspectives

The UI paradigm itself considered in the software architecture survey. Common views include: File, Console, Resource tree, package explorer, type hierarchy etc.. Common perspectives include Java, Resource etc..

12.1.4 Common operations and shortcuts

Ctrl + Shift + R : search and open a resource fast.
Well known: Format. Go to line. Undo, redo.

12.1.5 Functionalities

Various views provide nifty functionalities. We list some useful ones below.

12.1.5.1 Java

[**Incomplete**]

Part III

General purpose programming

13 C

13.0.6 Debugging with gdb

file reads in a source file.

13.0.6.1 Setting breakpoints

To set breakpoint: break mexFunction.
stop in, stop at: Stop in a particular function, stop at a particular line

13.0.6.2 Debugging after stopping

Once the debugger has stopped, can use: step, continue, exit etc.
where, whereami: Show the call stack, show the current location.
print: Display the value of a variable.

13.0.6.3 Memory allocation

Stack memory is often more limited compared to heap space (although both can be increased). int i[4000] uses stack memory - so it should be avoided.

13.0.6.4 Segmentation faults

Segmentation faults happen when the program tries to access out-of-bounds memory.
Dealing with mysterious errors (eg: segmentation faults) : sometimes even stuff printed to STDOUT and STDERR may not be printed properly in a bad crash, leading to ambiguity in pinpointing the source of the error. Commenting out code may then reveal it.

14 Simplified wrapper, interface generator (SWIG)

14.1 Purpose and mechanism

To use C or C++ code with python or java, swig is used.
One first writes an interface file describin the signature of the functions called, and of the objects passed.
Then, one runs a special program to automatically generate wrapper code (modules or classes) in the higher level language which provides functions and classes with similar signature, and which act as a bridge to the C code.

14.1.1 Persistence and memory

C++ objects persist between successive calls from the other language only if the code in the other language retains references to these objects.
Rather than keeping track of multiple such objects, it is simpler to keep track of a container object. So, the C code is often encapsulated within a class.
[**Incomplete**]

15 Python

15.1 Distinctive features of the language

Compared to C or Java, 1/3 to 1/5 code requried. Thence, higher productivity, software quality.
Good libraries. Can easily interact with C or Java code, and even extend classes specified there.
Program portability: Often compiled to bytecode, which is then interpreted into machine leanguage instructions.
Garbage collection.
Has Object oriented programming option; can also be used as a procedural or functional programming language.

15.1.0.1 Syntax conciseness

Clean and concise syntax: compare to perl; so more readable. As a general rule, python tries to use the same operators and function names for similar operations over different types of operands: Eg: +, len, dir.

15.1.1 Speed

An interpreted language. Speed is many times slower than C or Java. For vector operations, speed comparable with Matlab; but for looping, may be faster.

15.1.2 Linting

Python is an interpreted language - so many silly errors which would have been caught by a compiler are noticed only at run-time, if at all that code is run. So, Linting (and testings) is especially important.

15.1.2.1 Pylint commands

In code, one can say: # pylint: disable-msg=W0613

15.2 Writing, Building and executing code

Make .py files; begin with `#!/usr/bin/python`.

15.2.1 Important env variables

PYTHONHOME: location of the std libraries.
PYTHONPATH: default search path for modules/ package libraries, may refer to zipfiles containing pure Python modules (in either source or compiled form).
PYTHONSTARTUP.

15.2.2 Good IDE's

Eclipse with pydev. idle.

15.2.3 Point of entry

Can use interpreter. Or first line in file.py.
"python -c command [arg] ...", "python -m module [arg] ...", which executes the source file for module

15.2.3.1 Arguments

sys.argv, a list of strings has the script name and additional arguments from shell; an empty string if no argument is given.

15.2.4 Installing External libraries

Place a link in the site-packages directory.
Or run python setup.py build, python setup.py install.
Or do: sudo pip install pkgName or easy_install pkgName.

15.3 Help

help(object/ module)

15.4 Variables and data types

15.4.0.1 Matlab format

import scipy.io.
X = scipy.io.loadmat('mydata.mat'), scipy.io.savemat().

15.5 Interfacing with other languages, the OS

15.5.1 Numeric programming

numpy.

15.5.2 RPy or RPy2 for R

No easy plotting in sage.
from rpy import *. r.library ("..", `lib_loc =os.path.join(lib_path,"R"))`.

15.5.3 With Matlab

In sage use 'matlab.eval()'.

15.6 Other libraries

Regular expression: re.search(r'asdf', text) returns the first match object . m.group() returns the matching text.
re.findall returns a list of match objects. If regular expression contains pattern groups, tuple-list is returned.

16 Java

16.1 Distinctive features of the language

16.1.1 Purpose of design

Platform independence: so no direct access to memory.

Object orientated approach.
Memory management: Garbage collection.

16.1.2 Speed

Compiled into bytecode which can be run everywhere using native interpreters. Due to many optimizations - such as 'just in time' compilation, speed comparable with C for many applications.

16.2 Writing, Building and executing code

16.2.1 Good IDE's

Eclipse, netbeans.

16.2.2 Point of entry

SomeClass.main(String[] argVector).

16.2.3 Environment variables

CLASSPATH.

16.2.4 Building

Ant is a popular format/ tool for writing build-scripts for java.
Java packages are often exported as jar files; one then needs to add a jar file to the classpath. Within a jar file's manifest, one can specify a class which contains the main method, which will then be invoked upon using the command 'java -jar JARFILE'.

16.2.4.1 Decompiling

Java decompilers are also available.

16.2.4.2 Java Native Interface

Java determines where native libraries are to be found by looking at the environment variable `LD_LIBRARY_PATH`.

16.2.5 Help

Extensive javadocs, internet.

16.3 Data types, operators

Data are either objects or primitive data types.

16.3.1 Primitive data types

void, byte, char, int, double.

16.3.2 Objects

Even classes may be considered as special kinds of objects. All variables pointing to objects are actually references.

16.3.2.1 Object creation

variableName = new ClassName(arguments for the appropriate constructor);
Objects are created by calling the appropriate constructor: variable = new Object(..);.

16.3.3 Operators

Same as in C. '.' indicates membership or namespace.

16.4 Low level code structure

16.4.1 Sentence syntax

Code blocks and sentence syntax are same as in C.

16.4.1.1 Commenting

C style comments. Plus, /** javadoc documentation generation */.

16.4.2 Decision structures

Same as C.
The following is allowed Java 5 onwards: for (String arg : args) statementBlock.

16.5 Classes

Classes can be nested.

16.5.0.1 Referring to classes

When referring to classes, the namespace (Eg: packages.subpackage.Class) should be fully specified, unless there is a line in the file which says import packages.subpackage.*;.

16.5.1 Class members

A class can contain methods and properties: these are called class members.

16.5.1.1 Access modifiers

Scoping and access modifiers for class members are: public, private, protected, default (assumed when an access modifier is not specified explicitly), static, final (only for properties). Static members can be invoked without the creation of an object. Final properties are immutable.
Public members can be accessed and modified by code from other classes. Private members can be accessed only by code in the same class. Protected members can be accessed by code in the class and its descendants.

16.5.1.2 Properties

Syntax: AccessModifier dataType variableName (= optional initialization).

16.5.1.3 Methods

Syntax: AccessModifier returnType functionName(ArgumentList) CodeBlock. The method definition should contain an appropriate return statement: return XYZ;.
Invocation is same as in C. Arguments of Primitive datatypes are always passed by value, whereas object arguments are passed by reference.

16.5.2 Special methods

16.5.2.1 Constructor

A constructor is the method called by default when an object is created. If a constructor is not defined, a default constructor is used.
Constructors do not declare a return type explicitly. Otherwise, it is same as a function.
The constructor of the superclass is called before its own constructor is called. If a constructor calls a specific constructor of its superclass, it should do so in the first line.

16.5.2.2 Destructor

[**Incomplete**]

16.5.2.3 main function

A main function, being the point of entry for the code, should be declared as having the attributes: public static, and the return type void.

16.5.3 Important classes

16.5.3.1 Arrays

Arrays are actually objects of an Array class. Array indexing starts from 0, as in C. Example of an initialization

`{a,b,c}` specifies a 1 dimensional array with 3 members.
Multidimensional arrays in java are actually arrays of arrays: so var[0] and var[1] may be of different length.
Array indexing is done as in C.

16.5.3.2 String

Important methods include: substring, indexOf, startsWith, contains. split(regex).

16.5.3.3 Enumeration

```
public enum Day {
SUNDAY,MONDAY
}
Day day = Day.MONDAY;
switch(day) {
case MONDAY: ..
}
```

16.5.4 Important Superclasses, interfaces

16.5.4.1 Serializable

Implementing Serializable ensures that an object instance can be stored in a file and retrieved later.

16.5.4.2 Clonable

Implementing the Cloneable interface ensures that the traits of a given object can be copied using the copy() definition. ArrayList internally uses an array, but if copy were implemented such that only the reference to the internally used arraylist were copied, then both copies would be affected by any change in the arraylist. Hence, it is important to override copy() as appropriate.

16.6 Collection

16.6.1 Java Collections

There are interfaces $List < T >$, $Set < T >$, $Map < A, B >$. These are implemented in classes such as ArrayList, LinkedList, HashSet, HashMap.

16.6.1.1 Construction

Conversion from arrays can be done using: Arrays.asList().

16.6.1.2 Easy iteration

l.iterator() creates an iterator, but it need not be used explicitly.
This can be used with the for-loop:

```
for (Iterator<T>i = c.iterator(); i.hasNext(); ) {t = c.next(); BLOCK}

for (T t : c) BLOCK
```

16.6.2 High performance

Libraries like GNU Trove provide high performance native language libraries of collections which are faster and require less memory. They also enable storing java collections of primitive data types (traditionally only possible using wrappers like Integer and Float).

16.6.3 Important funcitons

retainAll(Collection), removeAll, containsAll, contains, remove.

16.7 Guava collections

These extend java collections to use higher order functions like scala's list.map.
THey follow the policy of 'when in doubt, leave it out'.

16.7.1 Mapping

```
List<Double>list2 = Lists.transform(list1,new Function<String,Double>()
{
public Double apply(String from) {
return Double.parseDouble(from);
}
});
```

16.7.2 Filtering

```
Iterable<String>filtered = Iterables.filter(list,new Predicate<String>()
{
public boolean apply(String input) {
return input == null || input.startsWith("B");
}
});
```

Similarly, Map.filterkeys can be used.

16.7.3 List-checking

A predicate called in is available: in(list1).apply('x').

16.7.4 Predicate building

16.7.4.1 Basic predicates

Basic predicates like

```
Predicates.equalTo
contains(Pattern)
```

is also available.

16.7.4.2 Predicate combination

These are boolean-valued functions acting on scalars. Their use in filtering and list checking is seen elsewhere.

`and(pred1,not(pred2))`

16.7.5 Function building

`forMap(Map<A,B>)` `compose(Function<B,C>,Function<A,B>)` `constant(T)` `identity()` `toStringFunction()`

16.7.6 Other collections

BiMap. SetMultimap. These lack public constructors, but a static create() method is available.

16.8 Exception handling

`try CODEBLOCK catch(ExceptionType e) CODEBLOCK` All exceptions descend from the Exception class: see javadoc for details.
Exceptions may be checked or unchecked - depending on whether the compiler forces the program to check for and handle exceptions thrown by a certain subprogram. Run-time exceptions usually go unchecked.
Exceptions of the type Error cannot be caught usually.

16.9 Multi-Threaded programming

The main thread always exists. Apart from that, other threads can be started. Extend the Thread class or implement Runnable interface. Threads can be started or interrupted. Threads can run or wait or sleep.
The synchronize keyword is used to enforce mutual exclusion from critical code blocks or methods.

16.10 Library functions

In general, search on the internet. Apache hosts many well maintained projects.

16.10.1 Mathematics

Colt and apache common math packages are widely used and maintained. The former is said to be more optimized : reportedly 2-3x slower than using fortran routines. Linear algebra operations on the latter reportedly seemed to be 6x slower.

16.10.2 Data structures

java.util has several classes, Eg: Set, List and their subclasses/ implementations.
Chief classes implementing the List interface: ArrayList (not thread-safe).

16.11 IO

Contained in the package java.io.

16.11.1 Stream

IO is handled using streams; which usually need to be opened and closed explicitly for efficiency.

16.11.1.1 OutputStream

Output streams can output bytes or readable text. Commonly used is PrintStream and PrintWriter (For writing characters rather than bytes); which include : println and print.

16.11.1.2 InputStream

[**Incomplete**]

16.11.1.3 Output to files

Initializing PrintStream or PrintWriter to write to a file can be painful because one needs to first initialize a FileOutputStream(fileName).

16.11.1.4 Standard output

System.in and System.out are always open.

16.11.2 File listing

java.io.File has useful functions: exists(fileName), list() to list contents in case it is a directory.

16.11.3 Read configuration files

Reading parameters from properties files, which store (key, value) pairs in lines with the format key= value is a very common task. So, there are specialized functions to deal with them: props = new java.util.Properties(); props.load(fileStream);.

16.11.4 Unit testing

Use junit.

16.12 Logging

General desiderata for a logging package is examined elsewhere.
Using logback (logback-core.jar and logback-classic.jar) with the slf4g interface package is attractive. Other options include log4j and java.util.logging, which may also be accessed using the generic slf4j interface (slf4j-api.jar).

16.12.1 Creating logger

Use LoggerFactory.getLogger(class).

16.12.2 Configuration

Create logback.xml file in the classpath, wherein logging level and appenders (destinations) with output pattern are specified.

16.13 CGI programming

16.13.1 Writing servlets

One must use the servlet API (preferably provided by the webserver being used) and create classes which extend HttpServlet, in which doGet and/or doPost methods are overriden.

16.13.1.1 Reading input

To decode HTML escaped characters, one can use the apache.commons.text package.
getParam, getParamMap.

16.13.1.2 Output

outStream object will have methods like setContentType, setEncoding, println.

16.13.2 Mapping to URL

One then maps a url to a servlet in the WEB-INF.xml file, using the `<servlet>` and other tags.

17 Scala

17.1 Distinctive features of the language

17.1.1 Java compatibility, similar properties

Scala bytecode is no different from Java bytecode. So, Scala shares the same properties as Java in terms of platform independence, speed, memory management, static typing etc..

17.1.2 Purpose of design

It allows us to do functional programming (creating anonymous functions, passing higher order functions) along with object oriented programming.

17.1.3 Conciseness

Scala is said to be significantly more concise than Java and slightly more concise than Python.
Scala's ability to pass function-objects make it concise to iterate over collections using foreach.

17.2 Writing, Building and executing code

17.2.1 Good IDE's

Plugins to Java IDE's.

17.2.2 Environment variables

SCALA_HOME pointing to the scala directory, other variables required by Java.

17.2.3 Building

Analogous to javac one has the scalac command.

17.2.3.1 Point of entry

main(args: Array[String]) method.

17.2.3.2 Interpreter

Using the scala interpreter, one can also run scripts or even individual commands typed in one at a time: but these are implicitly compiled to bytecode and run using the JVM.

17.2.4 Packaging

In order to run in an arbitrary machine, it is important for the scala-lib.jar to be packaged along with the project files.

17.2.5 Help

Extensive scaladocs, internet.

17.3 Data types

Scala is statically typed: even though it may only be made apparent by the value assigned to it. So, the compiler will not accept changes to the type of the variable by accepting values of different type.

17.3.1 Pure object-ness

Every value is an object. String values are represented as in Java. All operators are actual functions/ methods. But, this does not come at the cost of slower performance.

17.3.2 Variables, constants

var varName: Type = value. Constant: val valName = value. The ': Type' is optional.

17.3.2.1 Multiple assignment

val (x,y) = tuple
val Array(x, y) = array

17.3.2.2 Object instantiation

var varName = new className(argList). Sometimes, helpful factory methods are provided to avoid having to use 'new': Eg: List(), Array[Int].

17.3.2.3 Basic literals

Strings are enclosed in "". Chars are enclosed in ''. Int is written using numerals: 89.

Arithmetic operations 1/5 returns an integer: that is the way the /(int, int) function is defined! But, 1/5.toDouble returns the double 0.2.

17.3.2.4 Function objects

The type for a function argument is specified with: (argList)=¿returnType.

17.3.2.5 Implicit scope

Implicit scope is set for a variable by using the qualifier implicit in its declaration.

17.3.3 Type variables

type T = Char

17.3.3.1 Type objects

c.getClass or classOf[C]

17.3.3.2 Classes

Described in a separate section.

17.3.4 Type conversions

17.3.4.1 Explicit

obj.asInstanceOf[Type]. Or use an appropriate match statement.

17.3.4.2 Implict

There is syntactic sugar to allow implicit conversion of a value from type A to B in order to invoke a method x() defined for B but not on A.
To do this, one defines the class B with constructor B(val: A) and the new method x(). One then specifies the implicit conversion function: implicit def x(val: A) = new B(val).

From/ to java Such an implicit conversion is defined in a singleton object O, whose methods can be imported to the global name-space using 'import O._'.
scala.predef contains implicit conversions which are automatically imported.
import scala.collection.JavaConversions._ will import implicit conversions to and from several more java objects.

17.3.4.3 To Anonymous types

One can do this: implicit def x(i: Int) = new def <(str: String) = str.length < i
Doing this, one can easily overload the < function.

17.3.5 Important types

Classes important because of the type hierarchy are described elsewhere. Functions and collections are described in a separate sections.

17.3.5.1 String

This is actually same as java.lang.String. See various useful functions specified for java String.

Implicit conversions But, it is implicitly cast as List[Char], which enables the use of list methods. There are also implicit conversions which allow the following methods: +=.

Literals Apart from ”java strings”, one can use ”””literal string”””.

17.3.5.2 Regular expression

Syntax Creation from strings: `"""\w+\d+""".r`.
Subpatterns are specified using (). In case one wants to use () in the regular expression sense, arther than to specify a match group [As in M(r—s)], one uses: (?:).
regex1.pattern returns the java.util.regex.Pattern object.

Extract all matches regex1.findAllIn(str1) returns Iterator[Match].
Each Match object, which contains the group list, whose first element is the entire matched string, and substrings within it corresponding to subpatterns specified in regex1 appear next.

Replacement replaceAllIn(String, Match =¿ String)
replaceAllIn(String, regExpStr), where regExpStr is of the form `"$1 $2"`

Multi-pattern matching val regex1(var1, var2) = str1. This does string matching and grouping as defined by regex1 and assigns the matched groups to the var1 and var2.

Checking match regex1.pattern.matcher(str1).matches

17.4 Functions

17.4.1 Anonymous functions

(argList) => statementBlock . If only one argument is accepted, one can write: arg => statementBlock.
The syntax for specifying the argument list is described elsewhere.

If an anonymous function consists of one method application that takes a single argument, you need not explicitly name and specify the argument: so `print` can be an anonymous function, so we can write args.foreach(print).

17.4.1.1 Contextual conciseness

```
lst.foldLeft(10)(_ + _)
lst.map(1+)
```

17.4.1.2 Partial functions

Anonymous partial functions (with restricted domains as in mathematics) can be defined using case statements:
`{case(a,b) =>{..}}`
Its types and other methods are described in another subsection. The above is often used in maps.

17.4.2 Named functions

In the case of named functions, the only difference is that one writes def fnName [TypeList](argList): = statementBlock. Observe that => is not used here.
Named partial functions should be defined by explicitly implementing the PartialFunction trait.

17.4.3 Type List

The [TypeList], when present, declares a kind of hyper-class of functions whence objects are created as appropriate.
TypeList is comma separated, each element of which could either be a single literal, or something like A <: AnyRef : which means A which is subtype of AnyRef.
Among collections: $A[T] >: B[T]$ if $A >: B$.
However, even though it is desirable, $A[T1] >: B[T2]$ does not hold even if $A >: B$ and $T1 >: T2$.

17.4.4 Argument list

arg1: type1 = defaultValue1, arg2: type2 .. When the compiler can infer it, the types may be omitted.

17.4.4.1 Implicit arguments

If the sole argument to a function f(implicit arg) is prefixed with the keyword implicit, it will be provided automatically by the compiler when invoked `f` , provided a value of the appropriate type has been defined with the implicit scope: 'implicit val x = value'.

No explicit argument can be defined to occur after an implicit argument. [**Check**]Also, one can explicilty pass arguments - even if the parameters are declared to be implicit.
This especially is useful in order to define functions which make use of values implicitly defined by the run-time environment/ virtual machine (eg: objects of type Numeric[T]).
This facility is often used in using the type-class pattern.

17.4.4.2 Variable number of arguments

Use arg2: type2* for the last argument in the list.

17.4.5 Return value

By default the last expression evaluated in a function is returned. Return statements are of the form: return value.
'Unit' is akin to void in C.

17.4.5.1 Returning functions: shorthand

def fn(argList): (argList1) =¿ retVal1 statementBlock declares that fn returns a function with a certain signature.
Alternative notation for this is: def fn(argList)(argList1):retVal1 statement-Block.

17.4.6 Function type

The type of any function is specified using the notation: (argTypeList) =¿ returnType. This is syntactic sugar for the Function0 .. Function9 traits. Of these, the Function trait is an alias for Function0. One can define functions as objects with the trait FunctionN, but only one apply() function is then allowed.

17.4.6.1 Partial function

Function1 has a subtrait called PartialFunction which requires the isDefine-dAt() method to be specified. Note that this is distinct from partially applied function.

17.4.7 Partial application

var g = f(5, _:Int, _:Int).

17.4.8 Currying

One can always declare a curried function using the syntax for defining functions which return other functions. To convert an existing function to a curried fn, one can use Function.curried.

17.4.9 Composition

val fnComposed = fn1 compose fn2

17.5 Class

17.5.1 Definition

class className[Type](argList) { .. } creates a class. Both the Type and argList parameters are optional. When [Type] is provided, one is essentially declaring a hyperclass/ generic template, using which classes of various types can be referenced.

17.5.1.1 Constructor access

Default constructors are usually publicly accessible. But, one can specify access modifier private or protected: class className as modifierName(argList) .

17.5.1.2 Declare member variables with constructor

One can use the val or var keyword:
class ClassName(var var1:Type ..).

17.5.1.3 Nesting classes

It is possible to define one class within another.

17.5.2 Constructors and destructors

17.5.2.1 Main constructor

This creates an object, and sets all the fields and executes top level code contained in the class.

17.5.2.2 Auxiliary constructors

Define methods named this(..). This function should call the main constructor with appropriate arguments in the first line, using the statement: this(..).

17.5.2.3 Destructor

Define the override protected def finalize : Unit. Usually used to clean up files.

17.5.3 Methods

17.5.3.1 Define method

For syntax, see the section on functions. As a method is a member of an object of a class, one of its arguments is implicitly that object. So, it has the ability to modify the state of the corresponding object.

17.5.3.2 Without extra arguments

Argument list can be omitted in the definition. Eg: def a=expression or def a()=expression. Of course, the object is implicitly passed as an argument to them.
They can be invoked in a way similar to accessing attributes: using object.a - without ().

17.5.3.3 Dynamic method names

If a class extends a marker trait called Dynamic, one can define a special method called applyDynamic("meth", args). This method is invoked when one uses obj.arbitMethodName(args) or obj.arbitPropertyName.
This feature is very useful in defining domain specific languages.

17.5.3.4 Special methods

If the method name ends with _= as in fn_=, one can invoke it using objVar.fn = arg1.
Defining an apply function enables one to use the function notation to invoke it. Eg: obj() is same as invoking obj.apply().
Defining the unapply function enables one to define what happens when ClassName(val) is used as a possible value in a match-case statement.

17.5.4 Attributes

Arguments passed while constructing the class are always stored as attributes.
A class can contain various members which are variables and values: defined using the var and val keywords.

17.5.5 Access modifiers

For both attributes and methods: public, private. For methods: virtual.
These can be prepended to a member's definition, or one can assign it to several members by using a line like: 'public:'.

17.5.6 Hierarchy and inheritance

17.5.6.1 Inheritance

Inheritance from Java classes is possible.

To override a method, use the override keyword in method definition.
Clever syntax allows multiple inheritance.

17.5.6.2 Traits

Akin to an abstract class in C++, one can define Traits. In these one can define methods, unlike interfaces in Java. This is done using trait traitName { .. }.
Only Traits (not ordinary classes) can participate in mixins/ multiple inheritance.

17.5.6.3 Syntax

To extend a trait or class T1, one writes: class/ trait A extends T1. For multiple inheritance (aka mixin-class composition), one says
`extends T1 with T2 with T3`.

17.5.6.4 Mixins at runtime

One can also define classes inheriting various traits at the time of instantiation of objects (aka mixin various traits). To do this: new Dog with Exclamatory-Greeter .

17.5.6.5 Call superclass methods

Use super.f(). In case there is ambiguity due to multiple inheritance, use: super[AcestorClass].f.

17.5.6.6 Type hierarchy

At the bottom of the type hierarchy (ie they extend everything) are the classes Null (which is the descendent of any descendent of AnyRef), Nothing (which is the descendent of any descendent of Any). null is an object of type Null. It can be cast to almost any type.
At the top of type hierarchy is the Any class, whose subtypes AnyVal is the ancestor of all basic types (Character, Int etc..) and AnyRef is an ancestor to all classes and introspection methods.

17.5.7 Special classes

17.5.7.1 Singleton object

There are no 'static' members, so one has singleton objects.
object className .. instantiates a singleton object. If the class className has not already been defined, this defines it. If the class className has already been defined the singleton object thus created is called a 'companion object', which has special privileges like access to private members of other objects of the same class.

17.5.7.2 Case class

These are special classes objects of whose type may be fed to the 'match' function wherein code can be conveniently executed appropriate to the object's internal state.
Example definition: case class Number(value:Int).
Objects of these classes can be instantiated without using the new keyword.

17.5.7.3 Sealed class

Using the modifier 'sealed' ensures that the class cannot be inherited from except in the current file.

17.5.7.4 Abstract

Definition modifier: abstract. Ensures that the class cannot be instantiated.

17.5.7.5 Symbol

The Scala term 'mysym will invoke the constructor of the Symbol class in the following way: Symbol("mysym"). This is useful in defining domain specific languages.

17.6 Collections

Some collections are mutable (which does not fit the functional programming paradigm) while others are immutable. The principles of various general operations on lists are described elsewhere.
All collections are hyper-types: ie they require a type parameter in their definition: Collection[Type].

17.6.1 Creation

All collections have factory functions with a uniform syntax for their creation: besides using specific class constructors.
Eg: Traversable(1, 2, 3)
Iterable("x", "y", "z")
`Map("x" ->24,"y" ->25,"z" ->26)`
Array.fill(numItems)(fillerExpression)
Buffer.fill(numItems)(fillerExpression)

17.6.1.1 Numeric types

0 to 2 creates an iterator object. 0 until 5 produces (0, 1, 2, 3, 4). One can specify iteration steps by saying: 0 to -5 by -1.

17.6.1.2 Conversion from java

```
import scala.collection.JavaConverters._
javaCollection.asScala
scalaCollection.asJava
```

17.6.1.3 For expression

Aka for comprehension or for expression. It is not a for loop corresponding to the for looip of traditional languages. The difference is that it yields a value.
`for iteratorBlock statementBlock`. The purpose is that the statementBlock should be executed for values drawn from a certain iterator I, which may be defined in terms of values filtered and drawn from other multiple source iterators.
The iteratorBlock, which specifies the iterator whence values are drawn, is enclosed by () or {}. It specifies one or more values drawn from iterators - specified by $arg < -iter$ and filters/ guards specified by if(condition). The $< -$ states that arg is a value drawn from the source iterator iter. In addition, it may contain any other statement required for defining the values composing I.
By default, values are drawn fully from the iterator I before the statementBlock is executed - so its alterations of variables affecting the definition of the iteration block has no effect. But, this can be changed by using lazy evaluation function view (aka non-strict view) of iterators instead of the iterators themselves - iter.view.

17.6.2 Array

This corresponds to a Java Array, but is implicitly conversion/ wrapping to a mutable.WrappedArray, which extends Seq.

17.6.2.1 Data creation

a = Array.fill(size, value)

17.6.3 TraversableOnce

This is the ancestor of both Iterators and Traversable.

17.6.3.1 foreach

lst.foreach(fn). This does an action fn for each element in the iterator. This is equivalent to a for loop in Java or C.

Yield an iterator

17.6.3.2 Element indexing

find, indexOf.

17.6.4 Iterator

This extends TraversableOnce. The iterator is not a collection itself, but a way to access items in a collection one at a time - and has methods like next() and hasNext().
But, it is traversible once - so using functions like size, find etc.. irreversibly moves the index to the appropriate position.

17.6.4.1 Subiterators

take, drop. takeWhile, dropWhile return prefix and suffix iterators. slice returns an arbitrary slice. Filtering methods: filter, filterNot.

17.6.4.2 Growing

++, padTo, +, -.

17.6.5 Traversable.iterable

This trait extends TraversableOnce.

17.6.5.1 Conversions

Besides methods from TraversableOnce, there is 'iterator'.

17.6.5.2 Filtering

Also see TraversableOnce. In addition, withFilter fitlers elements in the collection does filtering in a lazy/ non-strict way to produce a Seq object.

17.6.5.3 Partitioning

splitAt, groupBy, partition. Note that these return an immutable map.

17.6.5.4 Element retrieval

Also see TraversableOnce. In addition: head, last.

17.6.5.5 Sublist retrieval, growing

Its methods for retrieving subcollections and for growing its size are same as in case of Iterator. In addition, it contains tail.

17.6.6 Seq

These extend Traversable.iterable. Those extending Seq are mutable and immutable.

17.6.6.1 Add elements

For mutable sequences: += adds an element.

17.6.6.2 Indexing, element access

They have the apply(), which enables accessing elements using the function-call syntax.
Other useful members: indices, indexWhere ..

17.6.6.3 Sorting, reversal

sortBy, reverse.

17.6.6.4 Comparisons

startsWith, endsWith, contains.

17.6.6.5 Multiset operations

intersect union diff distinct.

17.6.6.6 Conversion

seq.view gets a SeqView object, which ensures lazy/ non-strict evaluation of Seq-producing operations.

17.6.6.7 Iterative check

exists(booleanFn)

17.6.7 SeqView

This extends Seq, but all list/ iterator producing operations on it are done in a lazy/ non-strict way.

17.6.8 ArrayBuffer

This has the same semantics as an Array, but is more flexible and is iterable -as described in the data-structures survey. Also, it features constant-access-time.

17.6.8.1 Data creation

Use the constructor to set an initial size for the buffer. Then, use padTo(length, value) to increase the size of the array as necessary.

17.6.9 List

List extends Seq.immutable, and contain homogeneous elements. Any list method returns a new list, but does not modify the existing list.

17.6.9.1 List creation

Functions: List(valueList), Nil, val1::val2::Nil.

17.6.9.2 Add elements

Concatenation function: :::, insertion at the head: ::.

17.6.9.3 Drop elements

drop(2).

17.6.9.4 Get sublists

listVar(k). slice(a, b)

17.6.9.5 Iterative alteration

reverse. sort(binaryComparisonFn).

17.6.9.6 Removal

remove(checkerFunction).

17.6.9.7 Sorting

sorted, reverse.

17.6.9.8 String representation

`mkString("\n")`.

17.6.9.9 Option

A zero or one element List used in returning results from operations on collections. This is either a Some or a None object, the former of which has a value.

flatten Suppose that f(x) returns Option[T]. It is common to do var1.map(f), which returns List[Option[T]].
Some elements of this list may be None. In order to convert this to a List[T] while skipping over None objects, one can invoke flatten: var1.map(f).flatten. A shortcut is to say: var1.flatMap; described in general for Lists.

17.6.10 Set

The concept of a set is implemented as a pair of traits: mutable and immutable (located in packages scala.collection.(im)mutable).
These are extended by mutable and immutable HashSet classes. These are hyperclasses: so the type is HashSet[ElementType].

17.6.10.1 Adding elements

Use the += method which accepts multiple parameters to be added.

17.6.10.2 Membership check

contains()

17.6.11 Map

As in the case of a set, this is implemented as a pair of traits: mutable and immutable, and placed in different packages.
They are extended by mutable and immutable HashMaps, ListMaps. A HashMap is a hyperclass, so its type is HashMap[KeyType, ValueType].
Note that immutable maps, when return a copy of value objects to ensure that the value is not modified.

17.6.11.1 Creation

Map(tupleList) factory method. Iterator.toMap.
One simple way to create a map literal is: `Map("Dave" ->"Bass","Tony" ->"Guitar") -` this is enabled by implicit conversion of strings to a special object on which the `->` method is defined.

Traversable[Traversable] toMap if there are 2 traversables. Or do groupBy(Traversable =¿ keyElement).

17.6.11.2 Adding elements

The += method accept appropriate 2 element tuples as arguments.
map(key) = value or put(key, value).

17.6.11.3 Accessing elements

Can say map(key) or map.get(key) or getOrElse - get() does not throw an exception but returns an Option object.

17.6.12 Tuples

These are immutable and its members may be heterogeneous. This is a hyper-class, whose actual type depends on the values it holds. Eg: (1, ”df”) is Tuple2[Int, String].

17.6.12.1 Creation

var pair = (1, ”asdf”). `1 ->"asdf"`.

17.6.12.2 Accessing members

`pair._1`. Iteration can be done using the productIterator method.

17.7 Code structure

17.7.1 Sentence syntax

A statement ends in a newline or a semicolon. A block of statements is enclosed in {}.

17.7.2 Documentation

As in Java.

17.7.3 Mutliple assignment

val (var1, var2) = tuple1. val Array(var1, var2) = arr. Similarly one can unpack Seq, List etc..
Multiple assignment using regular expressions is described elsewhere.

17.7.4 Function invocation

f(val1, val2). fn(arg1 = val1, arg3 = val2).
fn will suffice if there is no argument.

17.7.4.1 Methods taking one arguments

`df format now` is same as df.format(now). This is the infix syntax, considering the containing object as one operand.
So, as everything is an object, 1 + 2 is same as (1).+(2).

17.7.5 Decision structures

while loops are as in java. if-else blocks are as in Java.
One can also use various methods and comprehensions defined for collection data types.

17.7.5.1 Replacing break and continue

There are no break and continue statements.
So, one can use: $lst.find(T => Boolean)$ to break a loop.

17.7.5.2 Switch case statements

testVar(s) match caseStatementList.
Note that one can match values of pairs of variables: (a, b).
A caseStatement is a partial function, eg: `case matchValue =>codeBlock`.
The default matching value is denoted by _.
Once a certain case is executed, other caseStatements are not executed.

Match value patterns Alternatively, one can match the data type: `case x:Color =>x`. One can also have patterns of the type: `val1 | val2 .. .`
One can also match state variables within an object of a 'case class': `case Number(1) ...`

17.8 Exception handling

trystatementBlock catch caseStatements. Each caseStatement in the catch block checks the exception type as in the case of the match function. Eg: case e:IllegalArgumentException =¿ e.printStackTrace() .
Exceptions are thrown as in Java.
Since Scala has no checked exceptions, Scala methods must be annotated with one or more throws annotations such that Java code can catch exceptions thrown by a Scala method eg: throws(classOf[IOException]).

18 Lua

18.1 Paradigms and extensibility

Multi-paradigm language. Functions can be treated as values, higher-order functions exist.
Highly extensible.

18.1.1 Building

Compiled to bytecode, executed.

18.1.2 Metatable

This can be used to effect inheritence.

```
fibs = { 1,1 }
setmetatable(fibs,{
__index = function(name,n)
--Call this function if fibs[n] does not exist.
name[n] = name[n -1] + name[n -2]
return name[n]
end
})
```

19 Perl

19.1 Distinctive features

This is an interpreted language. It is loosely typed.
It is very good at text processing.

19.1.1 Richness of syntax

It is very rich, so that there are multiple ways to accomplish the same thing; to the point where code written by one programmer may be unintelligible to another.

19.2 Running, building

19.2.1 Running

See perlrun on the internet. Commands to be interpreted are either entered in a special shell, or is passed in a file to the interpreter, or is passed in the command line.

19.2.2 Getting help

Use perldoc, perlsyn. Inbuilt function reference: perlfunc.

Part IV

Mathematical programming

20 Fundamental concerns

When doing mathematical computations - especially while using the floating point hardware representation, concerns dealing with overflow and underflow arise. These problems and various methods of dealing with them are described elsewhere.

21 Special libraries

21.1 Linear algebra libraries

21.1.1 Tuning to fit memory, processor

Matrix factorizations, matrix vector multiplications etc.. are tuned specially to the hardware, in order to achieve maximal speed-up. For example, they may break matrices up to fit the cache and combine the results.

21.1.2 Dense linear algebra

BLAS, LAPACK: fortran routines. clapack provides c interface to some lapack routines. atlas implements blas and some of lapack.

21.1.3 Sparse linear algebra

UMFPACK, SPOOLES. The standards for these computations is yet to be finalized, and it is an active area of engineering.

22 Matlab/ Octave

22.1 Introduction

22.1.1 Distinctive features of the language

Excellent plotting facilities; many libraries; interactive sessions good for exploring features of data; easy I/O: don't need to spend as much time in getting I/O right as in actual computation; excellent debugging facilities.
Octave is an open source clone of matlab.

22.1.1.1 Purpose of design

Matrix computations. This is meant to be a rapid prototyping language : It is not meant for fast implementations.

22.1.1.2 Versions

Certain OOP methods, properties don't work with 2007 version.

22.1.2 Speed

Matrix/ vector multiplications are implemented by calling efficient linear algebra implementations like LAPACK etc.. So, operations which can be specified using matrices/ vectors should never be specified using loops.
Also, certain functions like repmat, ones, sub2ind etc.. could be faster. Use packages like lightspeed which implement faster versions of these oft used functions.

22.1.2.1 Speeding up loops

When unavoidable, loops are known to be slow, even when efficient linear algebra routines are used internally. This is because of costs associated with interpreting each line of code in the loop at runtime.

Preallocate memory You can speed up loops by preallocating the memory used therein - that way matlab will not have to repeatedly allocate a new block of memory.

Use C Implementing the logic in C not only removes the overhead due to MATLAB interpretation; it also generally forces the programmer to be conscious of and avoid memory allocation overheads.
However, there is an additional cost involved in packaging arguments for use by the C function: So it is best not to call the same C function in a loop - rather it is better to code even the loop in C.

Make function logic inline This can avoid repeated memory allocation and overhead due to the function calling infrastructure and increase speed. BUT for unknown reasons, the reverse can be true sometimes!

22.1.3 Using External libraries

22.1.3.1 Locating libraries

Go to `http://www.mathworks.com/matlabcentral/`,
search, look under appropriate tags.

22.1.3.2 Installing and using

unzip package, add it to path: maybe can use addpath(genpath(dir)) to add subdirectories.

22.1.4 Building and executing code

Can write directly in matlab command window.
Or can write scripts (a sequence of commands in a file) or functions; Add directory to path; Use matlab to invoke.

22.1.4.1 Useful options

No GUI: matlab -nodesktop.
Command history: diary on. Others: addpath, savepath.

22.1.4.2 Good IDE's

Matlab.

22.1.5 Debugging

Can view matrices in tables, as an image etc..
Introducing break points: Use GUI or 'keyboard' can be inserted in the midst of any function. Then use dbquit or dbcont or dbstep or dbstack.

22.1.5.1 Profiling code for bottlenecks

profile on myfun; profile report

22.1.6 Help

help commandName.
For library documentation, see its docs.

22.2 Constants and data types

22.2.1 Data types of constants

Everything is an object. The most common object is an n-dimensional array.
boolean numbers are represented by 0 and 1.
str = 'stringValue': a character array.

22.2.2 Important constants

Inf, NaN.
isNaN(a).

22.2.3 Arrays

A = zeros(2,3,2,3) makes a 0 tensor of those dimensions.

22.2.3.1 Matrix

```
A = [2 1;
1 8]
```

; separates columns.
Block matrices: B = [A A+32; A+48 A+16].

22.2.3.2 Data generation functions

$-14 : 0.2 : 14$: Produces vector of equispaced numbers in the interval.

Basic constant arrays zeros, ones, trues, falses.
eye, speye.

Random array generation randn, random, unifrnd.

Sparse array generation spconvert(mat) returns sparse matrix with indices/ values specified in mat's columns. sparse().

Generate random permutations randperm(4) may return [2 3 1 4].

22.2.3.3 Array indexing

Array indexing is used to access a subarray in an array.

Indexing with scalars Array index starts at (1,1, ..). Shortcut for last element along any dimension: end. Eg: A(end, 5).

Indexing with index-vectors Entire row or col of an array can be selected using : symbol: A(:,3).

Indexing with one scalar Aka linear indexing
A(5) or A(v) are also meaning-ful, especially straight-forward for vector A.

Indexing with logical vector Eg: $A(A > 12)$.

Indexing tricks Indexing upper triangular portion of a matrix: create indices using triu(ones(m, n)).

22.2.4 Cell array

A = cell(m, n); creates a cell array of pointers (cells), usually pointing to matrices.

22.2.4.1 Cell elements: indexing, altering

A{i, j}: returns the contents of the appropriate cell in the array, whereas A(i, j) returns a cell (pointer).
Delete element from cell array: $cell(i) = []$.

22.2.4.2 Cell values

{23 34} creates a 1*2 cell array.

22.2.5 Data-type conversion

Use logical(), num2str(number, precision).

22.2.6 Structures

22.2.6.1 struct

s = struct('field1', values1, 'field2', values2, ...). isstruct.
isfield(s, 'fieldx'). rmfield.
get and put are abbreviated by syntax like s.field1.

Dimensions/ size of struct The size of the resulting structure is the same size as the value cell arrays or 1-by-1 if none of the values is a cell.

22.2.6.2 Use Java objects

```
mtype = java.util.Hashtable;
mtype.put ( 'numNodes',numNodes);
s = mtype.get('numNodes');
```

Note: Java heap space runs out of memory much more easily.

Checking emptyness mtype.get('asdf') returns [].

22.2.7 Function handles

Suppose fn f defined already. Then can use g = @f, can pass it to other fns just like variables.

22.2.7.1 Reparametrization

Can reparametrize functions while you are at it. Eg: If poly(x, b, c) is defined and b, c are specified, can use @(x)poly(x, b, c) to get a handle to a function with parameter x.

22.2.7.2 Anonymous functions

Can also create anonymous fn using an expression; Eg: $f = @(x)x^3 + x$.

22.2.7.3 Clearing memory

clear varName;

22.2.8 Objects

22.2.8.1 Object creation

c= ClassName(..)

22.2.8.2 Access/ manipulate objects

Use object.member.

Introspection methods properties(obj) or fieldnames(obj)
lists public properties; methods(obj) lists public methods. class(obj).

22.3 Memory, variable management

List variables in use: whos.

22.3.1 Clearing, recompiling

clear classes; rehash pathreset;. Need to do this if you alter class signature.

22.4 Data manipulation

22.4.1 Operators

22.4.1.1 Unary

transpose'; minus -.

22.4.1.2 Boolean valued operators

`< > <=  >=  == ~=`

22.4.1.3 Boolean operations on matrices

Boolean ops also work with matrices to make boolean matrices.
all(A) checks if $\forall i, j A_{i,j} \neq 0$.

22.4.1.4 Arithmatic operators

* + -

22.4.1.5 Multi-ary

Range specifiers: 1:5 1:2:5.

22.4.1.6 Component-wise operations

```
y = sin(x)./x;
```

22.4.1.7 Matrix operators

`A/B`: roughly AB^{-1}.
`A\B`: roughly $A^{-1}B$.

22.4.2 Operation with constant arrays: shortcuts

In A + 1: 1 is evaluated as a matrix of 1's.

22.4.3 Array processing functions

22.4.3.1 Array statistics

size, numel.
sum (along any dimension: 1 for sum along rows/ sum of columns, 2 for sum along columns/ row sum etc..).
max, min.
nz, norm, unique.
find: find non-0 elements of matrix.

Checking emptiness use isEmpty().

22.4.3.2 Reshape array

Delete row: A(2, :) = [].
If you store a value in an element outside of the array, the size increases to accommodate the newcomer.
Alter array shape: reshape.

22.4.3.3 Alter matrix

normr/c, tril, triu.

22.4.4 Vector processing functions

Vector statistics: histc().
dot(a, b) gets the dot product of a and b without having to worry about whether a and b are both column vectors in doing a'b.

22.4.4.1 String manipulation

c=['sdf' 'asdf'] or strcat() concatenates.
[firstToken reminder] = strtok(string, delimiters).
strrep : string replacement.

22.4.5 Matrix functionals

22.4.5.1 svd and eig

svd, svds (get only a few σ_i), eig, eigs.
eig and eigs can return vectors differing in signs. [**Find proof**]

Low rank approximations of symmetric A If A is symmetric, eigs is faster than doing svds, as it probably constructs $B = \left[\begin{smallmatrix} 0 & A \\ A^T & 0 \end{smallmatrix}\right]$ to compute svd. Can pass options which specify the structure of A (eg: symmetry), so that ew decomposition can be computed faster. Actually, $|(|\lambda) = \sigma$, but if ew's are close together, λ and σ returned by eigs and svds can be very different — but the resulting low rank approximations are both equally good.

22.5 Code structure

22.5.1 Sentence syntax

Sentence ends with ';' or new line: if former, execution is silent. Can continue writing in next sentence with '...'.

22.5.1.1 Commenting

% comment.

22.5.2 Functions and commands

22.5.2.1 Commands

Commands usually don't take vlaues/ variables, assume everything is a string etc..
Functions can be nested.

22.5.2.2 Functions without side-effects

Matlab functions are almost functions in the mathmatical sense: accept values and variables as arguments, return value. If they modify an object passed in the argument, they *must* return the object.

22.5.2.3 Syntax of function defn

```
function outputValue = fnName(arg1,arg2)
%  fnName(x,y)  function documentation
end
```

arg1 could be a function handle: you can then use arg1(..) to invoke the corresponding function.

To return multiple values, use function [a, b] = fnName(..).
To invoke functions from outside the file, they should be public: they should have same name as the file. Else, ye have a private function. Can't declare a fn inside a script.

22.5.2.4 Anonymous / temporary functions

```
f = inline('sqrt(x.^2+y.^2)','x','y');
f = @(x, y)sqrt(x.^2+y.^2);
```

Reparametrization is another way of defining temporary functions: this is described under the function handles section.

22.5.2.5 Nested functions

Definition Functions can be nested - by defining them as usual within a function body. But they cannot be nested within control structures.

Variable scope They can access and modify variables and utilize functions, if they are defined within functions defined at a higher level.
Variables defined within a nested function are not available outside.

Restrictions on modifying runtime workspace While debugging, if you break execution in a function in which a nested function is defined, you cannot modify the workspace by defining new variables. But you can define 'global' variables to get around this.

22.5.2.6 Checking input and output structure

Behaviour of a function can be made to vary based on the number of output variables expected by the caller, and the number of input variables required.

Variable inputs Use nargin to check number of arguments passed. Or, you can use structures to pass arguments.

Variable outputs Use nargout to check the number of output arguments expected. Then, add variable number of outputs like this: varargout(1) = op1; varargout(2) = op2; .

22.5.2.7 Invocation

Invocation of functions output = functionName(arg1, arg2);
`[m,n] = size(X)`. Strings must be enclosed with ".
Usually, reference is passed. Only the field being modified by the function will be passed 'by value'. If modified structure is returned as output, only changed fields will be altered upon function-return.

Asking functions for fewer outputs A function can return variable number (upto n) of outputs. But, when you invoke the function asking for k outputs, you cannot predict whether you get the first k of the n outputs or some other set - that is up to the function definition. Eg: qr().

To ignore return value `[~,n] = size(X)`.

Invocation of commands commandName arg1 arg2 . Arguments assumed to be strings: even if not enclosed by ''.

Invocation of methods Call superclass method: methodName@super(obj);

Invocation over multiple elements cellfun(fun, cell), arrfun(fun, array).

22.5.3 Define classes, packages

Base classes: value, handle.

```
classdef ClassName <package.superClass1 & superClass2
properties (Access = ..)
...
end
methods
% Constructor
function s = ClassName(..)
s = s@package.superClass1(...)
s = s@superClass2(...)
end
function delete(objRef)
...
end
end
events
EventName
end
end
```

In all non-static methods, the first arg is the object itself. object.fn(a, b) $\equiv$ fn(object, a, b).
No overloading functions within the same class.
If function modifies the object, it must return the object, like any fn in the mathematical sense!

22.5.3.1 Modifiers

Property modifiers: setAccess, getAccess = protected/ private/ public, Constant (not available in 2007 version), Static. Method modifiers: Access, Abstract = true, Static.
Property access methods: called when ye get or set property, before setting/ getting:

```
function value = get.PropertyName(object)
...
end
```

22.5.3.2 Directory structure

Under a directory in the path, +packageName dir, @ClassName dir or ClassName.m file. If a @ClassName folder is used, then ClassName.m containing the class definition should be placed within that folder.

mex files Packages can contain mex files:
invocation: packageName.functionName.

22.5.3.3 Ceveat about 2007 version

The modifier syntax (Static) can't be used in 2007 version: must use (Static = true): does not even give error message.
Bad error messages.

22.5.3.4 Check errors in definition

Try to construct an object or Use fieldnames on an object, or invoke a test method.

22.5.3.5 Specifying events

Make a subclass of handle. Broadcast event using notify(obj,'EventName').
Listen for events: addlistener(eventObj,'EventName',@myCallback).

22.5.4 Branching structures

```
if n == 0
T = t0;
elseif n == 1;
T = t1;
else
T = 3;
end

switch x
```

```
case {1,2}
disp('Probability = 20%');
case {3,4,5}
disp('Probability = 30%');
otherwise
disp('Probability = 50%');
end
```

22.5.5 Loop structures

```
for k= (Any *row* vector like 2:n)
T = [2*t1 0] -[0 0 t0];
t0 = t1;
t1 = T;
end

while booleanCondition
statements;
end
```

22.5.5.1 break and continue

Can break out of a loop or skip an iteration.

22.5.5.2 Avoiding loops in place of matrix ops

For efficient implementation, whenever matlab does large matrix operations, matlab uses PROPAC and other specially designed libraries to do things efficiently. So, if matlab has a matrix operation, always use that rather than writing your own loop.

22.5.6 Error handling

22.5.6.1 Throwing errors

```
ME = MException('component:mnemonic',...
'message');
throw(ME)
```

22.5.6.2 Throwing warnings

Use warning('msg') or warndlg(). These are displayed when you do warning('on').

22.5.6.3 Assertions about input

assert(condition, errorMessage) generates an exception with an error message if a condition is violated. This is useful.

22.6 Visualization

22.6.1 Creating a figure

figureHandle = figure() creates a figure.
Can later save this: saveas (figureHandle, name With Extension).
gcf returns handle to the current figure.
figureHandle = hgload(figure path) or open() loads a figure.

Set properties colormap gray;

22.6.2 Plotting data

The plotting functions accept suitable data, figure properties, return a figure handle.
subplot(m,n,p) breaks up the figure into m*n table, selects the pth cell for the current plot.
hist(vector, nbins) without output arguments. bar(vector).
plot (xAxis, yAxis, LineSpec, 'PropertyName', value, ...);
semilogy plots y axis in log scale.

22.6.2.1 Line specifications

It is composed of the following, in that order. Eg: '-+r'
Line styles: -, –, -., : .
Markers: + o * . x s d p h v̂ .
Colors: r g b c m y k g.

22.6.3 Plot properties

xlabel, ylabel, title.
grid on/ off/ minor.

22.6.4 Visualize matrix

22.6.5 Visualize matrices

spy a sparse matrix, for use with full matrices (aka non-sparse) use > 0.0001.
imagesc(A), image(A).

22.7 Prallel programming

22.7.1 Multi-threaded programming

Automatically enabled after 2008. Can be enabled with maxNumCompThreads = 2. Makes large matrix operations faster.

22.7.2 With matlab toolbox

Tightly integrated with many other mathworks toolboxes.

22.7.2.1 Processor pool

Worker processes can be enable on a cluster or on the local multicore machine : controlled by a distributed computing server.
Use: matlabpool open/ close/ size.

22.7.2.2 parfor loop

Operates like for loop, but there may be no variable dependency between loops - otherwise the program will fail to compile.

22.7.2.3 Other tools

Look in matlabcentral.

22.8 System IO

Use: [returnValue, commandOutput] = system(commandName);
To get environment variables: env(variableName).

22.9 Other library functions

Very rich in functions for matrix operations. Try to find pre-written functions before writing code from scratch.

22.9.1 Useful external toolboxes/ packages

software from Kevin Murphy and collaborators. Minka's lightspeed.

22.9.2 Solving Matrix problems

`A\b` solves system of equations Ax=b. lsqnonneg: Non negative least squares.
Fitting polynomials to x and y: polyfit.

22.9.3 File I/O

22.9.3.1 Log files

Can use file = fopen(fileName, mode); Mode can be w for overwrite, a for append ... fprintf(file, 'asdf'); fclose(file).

22.9.3.2 Loading formatted data

```
load someFile.mat;
save data.mat A1;
```

load can read ascii files with two space separated columns and make sense of it.

22.9.3.3 C like functions

fprintf, `fscanf(fid,"%d ...")`, fopen(fid, 'w'), fclose.

22.9.4 User interaction I/O

To print stuff: disp(). pause().
format long : set extra accuracy.

22.9.5 Solve general optimization problem

Described among optimization software.

22.9.6 Algebra

```
symadd('x^2 + 2*x + 3','3*x+5')
```

22.9.7 Measure time

tic toc.

22.10 mex: Using C functions

Write c files for efficiency.

22.10.1 Interface

```
#include "mex.h"
void mexFunction(int numOutputArrays,mxArray *pOutput[],
int numInputArrays,const mxArray *pInput[])
...
```

Compile them in matlab prompt: mex fileName.

22.10.1.1 Debugging

See tips online.

22.10.2 mxarray datatype

This includes the array's type, matlab variable name, dimensions. Non-sparse arrays contain pr and pi to store the real and imaginary parts of the data: these are one dimensional arrays.

22.10.2.1 Creation

Use functions like: mxCreateNumericArray, mxCreateCellArray, mxCreateCharArray.

```
mxArray *myarray = mxCreateNumericArray(..);
mxCreateDoubleMatrix(1,3,mxREAL);
```

22.10.2.2 Array access

mxGetPr, mxGetPi, mxGetData, mxGetCell.
mxGetM, mxGetN: get matrix dimensions.

22.10.2.3 Array modification

mxSetPr, mxSetPi, mxSetData, mxSetField. Can use memcpy to copy arrays at one shot.

22.10.2.4 Memory management

mxMalloc, mxCalloc, mxFree, mexMakeMemoryPersistent, mexAtExit, mxDestroyArray, memcpy

22.10.3 Executing matlab commands

Both mexCallMATLAB and mexEvalString execute MATLAB commands. Use mexCallMATLAB for returning results (left-hand side arguments) back to the MEX-file. The mexEvalString function cannot return values to the MEX-file.

22.10.4 User IO

mexPrintf, mexWarnMsgTxt, mexErrMsgTxt.

22.10.5 Input validation

mxIsChar, mxIsClass(prhs[0], "sparse"), mxIsComplex.

22.11 Java

22.11.1 Advantages

Using loop operations can be 2x faster in java. For matrix operations, the fortran routines used by matlab are faster; but using java libraries like colt is said to be not too much worse.
Also, Arrays and values are automatically converted between matlab and java formats.

22.11.2 Classpath

javaclasspath shows static and dynamic portions of class path. The former is loaded from [matlabroot `'\toolbox\local\classpath.txt'`] . The latter is set using javaclasspath or javaaddpath. The class path is refreshed using [clear java].

23 Optimization software

23.1 cvx

23.1.1 Distinctive features

23.1.1.1 Disciplined convex programming framework

Makes specification easy. Abstracts away the mathematical and methodological details (like what underlying solver to call?).
Tries to mimic the way people formulate convex programs: drawing from a library of convex functions, combining them in various valid ways to preserve convexity. Similarly, it has a rich, expandable library of functions with known properties, plus it specifies valid ways of composing and combining them.
Good way to check convexity of the problem!

23.1.1.2 Defects

A generic solver. Not suitable for large problems.

23.1.1.3 Other features

It is closely integrated with matlab.
Possible to specify the underlying solver used.

23.1.2 Help

See cvx manual on website, many examples, quickstart file.

23.1.3 Specification structure

```
cvx_begin
    variable p(numParams);
    minimize norm(A*p − ones(numRows,1), Inf);
    [[or minimize(a + b), not minimize a + b]]
    subject to
    p >= 0;
    p <= 1;
cvx_end

Other modes:
cvx_begin sdp : interprets matrix inequalities as conic.
cvx_begin gp
```

All constraints and optimization function for minimization should be convex - else, cvx will reject the problem and try to provide a helpful message trying to explain the reason for the rejection.

23.1.3.1 Matrix variables and constraints

Multiple constants/ variables are specified using matrices.

```
Matrix variables:
variable X(n,n) symmetric;

Semidefiniteness constraints:
X == semidefinite(n);
```

23.1.4 Error reporting and dimensions

To stop verbosity: `cvx_quiet(true)`.

23.1.4.1 Scalars and 1*1 matrix

If you try to add a scalar and a 1*1 matrix lke a^Tb, cvx fails with a cryptic error message.

Assuming dimensions of variables automatically If you declare variable b intending for it to be a scalar; but then try to add it to a vector in the constraints, cvx does not report an error.

23.1.5 Solution, program variable

23.1.5.1 Datatype of the program variable

The program variable is different from matlab variables. While the optimization is underway, p is a cvx object. After optimization is done, p is the optimal solution - just a normal matlab variable.

23.1.5.2 Solution variables

opt_val, tolerance etc..

23.2 Matlab optimization toolbox

Built into matlab.

23.2.1 Unconstrained minimization

23.2.1.1 fminsearch

Solves unconstrained optimization problem, where objective is assumed to be continuous. fminsearch can often handle discontinuity, particularly if it does not occur near the solution.
fminsearch uses the simplex search method. This is a direct search method that does not use numerical or analytic gradients as in fminunc.

23.2.1.2 fminunc

fminunc does the same thing, except it cannot handle discontinuities. [**Check**]

23.2.2 fmincon

Finds a local minimum of a constrained nonlinear multivariable continuous function. Solves $\min f(x) : c(x) \leq 0; c_{nl}(x) \leq 0; Ax \leq b; A'x = b'; l \leq x \leq u$.
It uses finite difference approximation of the gradient when the gradient is not available.

```
[x,fval,exitflag,output,lambda,grad,hessian] = fmincon(
    fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,options,P1,P2, ...)
```

Parameters to ignore can be specified by passing [].

23.2.3 Other functions

fminbnd : minimizes $f : R \to R$.

23.2.4 Options

A structure with options.

```
options = optimset('Display','iter','TolFun',1e-2);
options = optimset(options,'TolX',1e-2);
```

24 R

24.1 Introduction and use

24.1.1 When to use

Statistics is the emphasis, not matrix manipulation. It is an expression language.

24.1.2 When not to use

Text processing and general purpose programming are painful.

24.1.3 Writing, Building and executing code

24.1.3.1 Script execution

source("/path/file.R").

24.1.3.2 Reloading changed code

library("R.utils");
sourceDirectory("work", modifiedOnly=TRUE,
pattern="[A-Z]*[.]R$", recursive=FALSE);

24.1.3.3 Important environment variables

`R_LIBS`: the place where R libraries are installed.

24.1.3.4 Working environment

getwd(), setwd(dir)
options(): Example options(digits=3)

24.1.3.5 Command history

history(), savehistory, loadhistory.
Can use Ctrl+R as in BASH.

24.1.3.6 Good IDE's

Native GUI Graphical data entry: data.entry(x), edit(x).

24.1.4 Listing and memory

objects() lists objects.
rm(objA, objB) removes certain objects from memory. rm(list=ls()) removes all objects.
exists() checks for the existance of an object.

24.1.5 Development environment

RStudio is good. Interactivity with RapidMiner is useful.

24.1.6 Debugging

The following can be inserted in the midst of code:
browser(): breaks execution and allows debugging with arbitrary code: like keyboard in matlab. Use cont to continue the program. Q halts execution.
debug(): marks function for debugging.
trace() function modifies a function to allow debug code to be temporarily inserted.
setBreakpoint("fun.R#20")
traceback() is useful when error is encountered.

24.1.7 Help

help(command), ?command,
apropos("log"), help.search("log"), example(commandName).

24.1.8 libraries

24.1.8.1 Seeking

help.search(fnName);

24.1.8.2 Using

24.1.8.3 Installing

Many packages are listed in cran internet repository.
chooseCRANmirror()
install.packages("igraph"). For usage, see the packages subsection.

24.2 Data: types, values, variables

24.2.1 Names and namespace

24.2.1.1 Valid names

a.b and a_b are valid name; but by doing so, you are not creating a structure named a. A name can start with '.', but if so it cannot be followed by a digit.

24.2.1.2 Namespace

Uses: avoid name conflicts, structure code well.
To call a function b in a namespace a, use: a::b(). If b is hidden, then use a:::b().

24.2.1.3 Accessing list members directly

Use attach(Lst). detach() reverses this.

24.2.2 Object

Everything is actually an object with attributes.

24.2.2.1 Accessing attributes

attrName(objName)

24.2.2.2 Setting attributes

attr(object, attribute) is used to get or set an attribute.
Or one can use structure(object, attr=value).

24.2.2.3 Important attributes

mode: details about the type of data contained.
length.

24.2.3 Modes and type

Mode: details about the type of data contained. This is distinct from the type of the object itself (data.frame, or vector ..). So, type is akin to a generic/ template/ meta class in Java made concrete by specification of mode arguments corresponding to contents.

24.2.3.1 Basic modes

Boolean. Numeric. Integer.

24.2.4 Scalar values and operations

24.2.4.1 Special values

TRUE, FALSE. Inf. NA: 'not available' or missing values. NaN is also a special case of NA.
These can be checked using is.na() and isNaN functions.

24.2.5 Data conversion

as.array(), as.data.frame etc..
Use methods(as) for a list of such methods.

24.2.6 Vectors

24.2.6.1 The basic math object

A scalar is actually a vector with one element in R. An array is also internally a vector. A string is a character array.

24.2.6.2 Homegeneity

Vectors are homogenous: their 'mode' attribute is character, numeric, logical.

24.2.6.3 Named entries

names(x) = stringList is a way of giving names to indices.

24.2.6.4 Indexing

A vector can be indexed with a vector of a] logical elements, b] positive integers, c] negative integers.
Eg: x[1:10] picks 1st 10 elements. x[-2]: all but x[2].
x['abc'] is valid if entries are named.
tail(x, k) picks the k last elements.
To append to a vector: append(v, val)

24.2.6.5 Creating vectors

c(1,2) : Using the concatenation function.
seq(); This can be abbreviated using the : operator.
rep(vector, timesToRepeat) is the replicator function.
vector() creates an empty vector.

24.2.6.6 Vector statistics functions

length, max, min, sum, mean, median, cummax, cummin, cumsum, cumprod, range, prod. cor: correlation.
which: gather 'TRUE' values from boolean vector.

24.2.6.7 Apply scalar functions

lapply returns a list. sapply, a simpler wrapper around lapply, returns a vector by default. vapply is an
apply returns an array or a vector but acts on arrays.

24.2.7 Factor

Factors are ways of storing a label-vector. They can be created using factor(labelColumn).
An important attribute is levels, which contains the set of labels used.

24.2.8 Strings

These are character vectors. Enclosed in ”” for brevity.

24.2.8.1 String manipulation

concatenation: paste(vector). substr gets a substring.
strsplit(v, sep) returns a list, having split each element in v.

24.2.9 Dates

24.2.9.1 String connection

`d<-as.Date("1995/12/01",format="%Y/%m/%d")` converts string to date.
format(dt) converts date to string.

24.2.9.2 Arithmetic operations

Then, one can add days with d+20.

24.2.10 Arrays/ matrices

24.2.10.1 As special vectors

Arrays are just vectors which support multiple subscript indexing. So all operations and restrictions that apply to vectors apply to arrays.
Arrays are stored column by column.
The dimensionality is stored as a vector in the dim attribute.

24.2.10.2 Indexing

A[a, ...]. Any subscript can be replaced with the sort of vector used to index vectors. If a subscript is omitted all values in that dimension are chosen.
Eg: A[,c(3:5)]: picks 3 cols.

24.2.10.3 Data creation

array(vector, dimensions). Or just set the dim attribute after creating a vector.
Or concatenate various vectors or arrays: c(v1, v2)

Matrices x = matrix(vector, nrow = 3). matrix(0,nrow=n,ncol=n)
Row or column binding functions: rbind(vec1, vec2); cbind.

24.2.10.4 Dimension-wise oeprations

apply applies functions to margins of an array; apply(x,1,max): gives row max.
sweep(m, 2, colSums(m), FUN=”/”)

24.2.10.5 Matrix functions

t(A): transpose.
%*%, %$\hat{\%}$: matrix mult and exp.
diag(A) extracts diagonal, creates a diagonal matrix depending on the argument.
sum, rowSums, colSums, rowMeans, colMeans. cor: correlation.

Linear algebra eigen, svd, qr. solve(A, b). lsfit(A, b).

24.2.11 Lists

Lists are heterogenous. They are a combination of a Hashmap and a list. They are very convenient to use as structures.

24.2.11.1 List creation

$L < -list(1, a, b)$ or $L < -list(a1 = 1, a2 = a, xyz = b)$ for named lists.

24.2.11.2 Accessing list items

L[[4]], L[["fieldName"]], L$fieldName. While indexing, list names can be abbreviated: Eg: 'cov' instead of 'covar', as long as the interpreter is still able to uniquely identify the intended member.
TO see if a member exists, use: fieldName %in% colnames(Lst).

24.2.11.3 Concatenation

c(L1, L2) returns a single list with members from both.

24.2.12 Data frame

Data frames are the R concept for data tables or matrices which can consist of columns of mixed types which can also have a name.

24.2.12.1 Creation

They are often read in from files - using read.csv for example.
By concatenating vectors: df = data.frame(n, s, b).

24.2.12.2 Conversion

Or from a matrix: data.frame(A). It can be reconverted to a matrix using data.matrix.

24.2.12.3 Indexing

Indexing is done as in the case of two dimensional arrays. If column headers exist, they can be used for indexing.

```
drops <-c("x","z")
DF[,!(names(DF) %in% drops)]
DF$colName
```

24.2.12.4 Searching

`which(sbux.df$Date==` ”3/1/1994”)

24.2.12.5 Concatenation

As in the case of matrices

24.2.13 Model Formulae

$\sim$ operator is used to separate left and right sides of formulae.
Syntax: response $\sim$ predictor variables (separated by +).

24.2.14 Functions

Functions are actually objects.

24.2.14.1 Definition

```
my.mean <-function(x1 = defaultValue,y1) codeBlock
```

The value returned is the value evaluated by the last expression in the function definition. Multiple return values are usually handled using lists.

24.2.14.2 Variables used

Usually variables have a local scope: they cannot be accessed outside a function. They can use variables from the parent scope, say g. The value of g is bound from the parent scope. If a normal assignment is made to g: $g < -0$, g is then taken to refer a local variable. If g is not bound to a value, but is required in the function definition, there is an error; as it is not bound either in the definition or by an argument. However, if $g << -0$ is used, then the parent-scope variable is updated, and g acts as the ’state’ of the function.

24.2.14.3 Operator defintions

The function name can be replaced by %*% for example.

24.2.14.4 Invocation

Invocation can be done as in C, using a sequence of values. Together with positionally specified values, one can pass named arguments in any order. Eg: f(3, a=1, b = 2) or f(a=1).

24.3 Operator

24.3.1 Assignment

$< -, - >$, assign(). $<< -$ is used to make global assignments: assignment to a variable outside the local scope.

24.3.2 Scalar operators

24.3.2.1 Arithmetic operators

As in C.

24.3.2.2 Operators on booleans

`|,&,||,&&`, where the latter result in 'short-circuit' evaluation, where the second argument is evaluated only if necessary.

24.3.2.3 Operators which produce booleans

`>,<,==`.

24.3.3 General Vector operators

24.3.3.1 Mapping over elements

lapply can be used to apply scalar functions to vector elements, while apply() can be used to apply vector functions on array rows.

24.3.3.2 Arithmetic and boolean ops

All scalar arithmetic operators are extended to be meaningful when provided vector arguments: even when they are not of the same size.

24.3.3.3 Lengthening of arguments

All shorter arguments are extended by repetition to have the size of the longest vector: Thus 1+c(2 3) is define.

Examples c(1,2,3,4)/c(4,3,2,1). c(1,2,3,4) + c(4,3) yields 5 5 5 8.

24.3.3.4 ifelse op

ifelse operation uses a logical vector as a condition.

24.3.3.5 Set membership

```
drops <-c("x","z")
DF[,!(names(DF) %in% drops)]
```

24.3.3.6 Missing value identification

is.na(x) returns true for both NA and NAN values. Note that this is different from the syntactically undecidable expression x == NA.

24.4 Code structure

Every line of code is an expression or a sentence.

24.4.1 Code blocks

{} encloses code block.

24.4.2 Sentence syntax

Sentences end with newline or ;.
# comments.

24.4.3 Decision structures

if(..) codeBlock; optionally followed by else codeBlock. for(var in vector) codeBlock. while(cond) codeBlock.

24.4.4 Packages

To be able to use a package, one says: library(packageName). Standard packages are automatically available. Note that this is distinct from the idea of a namespace.

24.4.5 Organization with lists

Can group functions in lists which are declared in separate files. These files are then (re)loaded using commands like source or sourceDirectory.

24.5 IO

24.5.1 File I/O

To load data from a table, use read.table(), read.csv (fileName, header=FALSE, stringsAsFactors = FALSE), write.csv(x, file).
Write in matlab format: library(R.matlab), writeMat(filename, A=mat).

24.5.2 User interaction I/O

24.5.2.1 output

a+b prints a value, which is then lost.
print. printf is available in the base package.
sink('fileName') diverts output to a file. sink() restores it to STDIO.

24.5.2.2 Input

x= scan(): keyboard input, no commas.

24.5.3 Plotting and tables

`plot(y ~ x)` produces a scatter plot.
Tables can be produced with `xtabs(y ~ x)`.

24.6 Data preparation and exploration

scale(x, center = TRUE, scale = TRUE) normalizes columns using the mean and standard deviation.
Getting covariance matrix: cor(x, y = NULL, use = "pairwise.complete.obs")

24.7 Modeling

Several useful functions are provided to evaluate fitted models in the package stats, which is loaded by default.

24.7.1 Classification

Decision tree learning: rpart

24.7.1.1 Logistic regression

```
glm(model,data=tblName,family = binomial())
ret <-glm.fit(x=X,y=z,family=binomial())
```

Options for family include: binomial(link=logit)
The return value is a list which includes coefficients and fitted.values.

24.7.1.2 Logistic regression with l1/l2 regularization

Use the glmnet package (requires gfortran).
Example:

```
returnList <-cv.glmnet(X,y,family = "binomial");
```

returnList contains the following vectors: lambda (corresponding to the lagrangian multiplier for the l1-norm), cvm - the corresponding mean cross-validated errors, glmnet.fit: the fit weights.

24.7.1.3 Decision tree: rpart and tree

```
ret <-rpart(model,tblRet)
print(ret)
```

24.8 Other library functions

24.8.1 Random sampling

Sampling from distributions: runif. sample(x, size, [replace=TRUE])

24.9 Write C extensions

24.9.1 Write C code

Useful libraries: #include ¡R.h¿ #include ¡Rmath.h¿
Signature: void getSamples(int *input, int *output)

24.9.2 Calling C code

Compile C code: R CMD SHLIB foo.c
Loading: dyn.load(”foo.so”)
Calling: .C(”foo”, n=as.integer(5), x=as.double(rnorm(5)))
This returns a list of return.

25 Rapidminer

25.1 Introduction

25.1.1 Purpose

Rapidminer provides convenient tools for data loading, exploration, modelling, visualization.

25.1.2 UI and API

It comes with a nice GUI. It provides Java API. It also has a scripting interface.

25.1.3 Data types

(Bi or Poly) Nominal, Integer, Double, Varchar.

25.2 Processes

One can visualize the data mining process as a tree of operators. The leaf nodes correspond to the result nodes. Processes can be edited in the design view : here it looks like an electronic circuit. This is effectively programming using a GUI.

25.2.1 Operators

Operators are grouped under various categories: like Modeling, Data transformation, repository access, evaluation/ validation, process control (looping, conditions). These groups contains various subgroups.
Each operator has an input and output, and may require additional settings.
An operator may be nested: itself composed of sub-operators.

25.3 Data view

The data view has different tabs to show meta-data, the actual data, a good interface for plotting.
The meta-data view shows useful summary like mean and standard deviation for numeric data and, mode /min classes for nominal data.

26 Spreadsheet programs

26.1 Google docs

26.1.1 Cell referencing

Columns are numbered with letters and rows with numbers. A cell can be specified using a latter and a number.
When cell references are copied (perhaps as part of formulae), the references may be automatically changed in a certain way - this is called relative reference: Eg: The reference A20 used in cell A2, when copied to B2 will automatically be changed to B20.
When such change does not happen, we have absolute reference to a cell. Eg A\$20, and \$A20 (the former indicating that the row is fixed and the latter fixing the column), \$A\$20.

26.1.2 Range specification

A:A, A2:A30.

26.1.3 Filtering

26.1.3.1 Condition specification

"¿0", value.

26.1.4 Counting

COUNTIF(rangeToCheck, condition, rangeToCount). rangeToCount is optional.
COUNTBLANK. COUNT.

26.1.5 Summation

SUMIF, SUM. These have syntax similar to countif and count.

26.1.6 Averaging

AVG(RANGE)

26.1.7 Looking up values

vlookup()

Part V

Shell Scripting and OS work

27 General tasks

Web browsing is described elsewhere.

27.1 Scanning documents with camera

The following 3 steps can be accomplished online on picasa's piknik editor.

27.1.1 Cropping

The image may need to be cropped to the interesting region.

27.1.2 Enhancement

When the camera captures the image of a document, because of inadequate light, there is a shift in the white-content of colors captured - eg: white becomes a shade of grey. So, processing the camera picture to increase brightness and contrast of the image (aka Enhancing) is required. This ensures that, when reprinted, the colors appear closer to the original - ie they are not darkened.

27.1.3 Compression

For efficient storage and transmission, the image may need to be compressed. The color range and image size may be reduced. 65% quality JPEG compression is usually adequate.

27.2 File synchronization

Dropbox service syncs files across computers; but it does not preserve deleted files for more than a month.

27.3 VOIP

Skype, google chat.

27.4 Booting to special modes

WIndows safe mode: Press ctrl during startup.
Jawbone headset pairing mode: Turn on while holding speak button.

27.4.1 Atrix boot modes

27.4.1.1 Fastboot

Turn on while pressing volume-down button until you see 'fastboot' on the top. Pressing the down button shows various other options, including "Early USB Enumeration". up button selects an option. Pressing the up button upon seeing 'fastboot' selects the fastboot mode.
you only get about 1-2min to do anything before the phone reboots on its own. [**Check**]

27.4.1.2 Recovery

One can enter the recovery mode using adb reboot recovery. Thence, udpates may be manually installed.

27.5 Virtual machines, emulators

An emulator provides the same interface as another operating system to some application. A virtual machine simulates computer hardware.

27.5.1 Emulators on linux

WINE emulates windows.
mono emulates standardized part of the .Net framework. Command: mono something.exe.

27.5.1.1 Java virtual machines

[**Incomplete**]

28 Pocket computer OS usage

28.1 Special tasks

28.1.1 Syncing media

This sometimes involves more than merely copying files. Special directory structures, thumbnails may be created.
For syncing media: banshee on ubuntu, windows itunes for apple devices.

28.1.2 Podcasts

Mediafly is a good podcast listening service - one can store list of channels/ podcasts online; one can arrange to preload them when access to WiFi is available.

28.1.3 Scanning

Genius scan uses camera and does necessary post-processing to increase image brightness and contrast.

28.1.4 Offline navigation

Applications are available to create maps from online sources for offline use on the pocket device. Eg: mobile atlas creator.

28.1.5 Ebook conversion

28.1.5.1 Calibre

ebook-convert a.html a.mobi . This reads and uses creates table of content in a.html.

ebook-viewer
web2disk URL

28.1.6 Keypad commands

Pocket computers equipped with cell-phone radios often have diagnostic/ system information routines accessible through dialing special codes.

28.1.6.1 Android

`*##*checkin#*##*`Phones home to check for updates.
`*##*info#*##*`- Enters a detailed phone information menu. Lets you turn off cell phone radio (while enabling bluetooth).
`*##*1472365#*##*`: Access to the GPS config menu.
More info here

28.1.7 Unlock bootloader

The bootloader may be replaced - making rooting easier.

28.1.7.1 Atrix running Gingerbread

The procedure is as follows.
Install android sdk
Reboot phone in fastboot mode.
Connect to computer.
Type fastboot oem unlock
[**Incomplete**]

28.1.8 Exploring files from a computer

Usually, when connected to another computer, one can browse and manipulate only a subset of files on the pocket computer - this makes file operations safer and avoids 'bricking' or complete software failure.

28.1.8.1 Android

One enables usb connections in debugging mode on the pocket computer, installs the android sdk, does sudo adb start-server, and then sudo adb shell etc..

28.1.9 Enable and use root access

This allows user programs the ability to gain root access - a feature usually disabled by default.
To use root from adb, just type su.

28.1.9.1 Atrix running gingerbread

Enter the fastboot mode, copy certain files, reboot and gain access using the sudo ./adb shell command; then replace su, SuperUser.apk + some other stuff (see ??).

28.1.10 File transfer: Android

28.1.10.1 Enable Writeability

adb shell. adb mount - gives mount points like /dev/block/mmcblk0p12 /system . Then one does: mount -o remount,rw /system
If the above fails, one can transfer to /sdcard-ext and then move the file to any location under superuser mode.

28.2 Kindle keyboard

28.2.1 Diagnostic pages

411 page: Alt + RQQ. 611 page: Alt + YQQ.

28.2.2 Shortcuts

Alt + F: next song. Pan: alt + arrows. Text to speech: Alt+Sym. Numbers: Alt + qwerty.

29 Linux OS

Map tasks and commands.

29.0.2.1 perl

Running perl from the command line is a very popular option. Various switches are described in perlrun.
-e expr executes expressions provided.
-n ensures that the expression runs on each line from file specified either in the command line or passed in stdin.
-p Same as -n, except output of each line is printed.
-a ensures that each line is automatically split to produce a token array (whose name may be arbitrary).

29.1 Graphical user interface

GUI software stack is described elsewhere.

29.1.1 Common settings

All possible sessions are represented by a file in a certain location for the login-session manager to find.
All available applications are listed in a single location.

29.1.2 Gnome

After changes, gnome can be restarted by entering 'r' in the 'Run command' interface - usually invokable by Alt+F2.

29.1.2.1 Editing configuration

gconf-editor provides a good 'registry'-like interface to all gnome applications'/ desktop environment settings.
'System settings' and ubuntu tweaker provide more polished but limited GUI's.

29.1.3 KDE

General settings are edited using systemsettings.

29.2 Others

xfce was tried successfully with Ubuntu 11.10.

29.3 Editing

29.3.1 Editors

Useful features include: copying/ pasting, automatic indentation, ability to easily (un)comment, brackets which auto-close.

- general: kate.
- bib editor: jabref.
- html editor: bluefish, kompozer.
- latex editor: kile. texmaker for indic script.

29.3.2 Programmatic editing

Pipe file list to `|xargs perl -w -i -p -e "s/str1/str2/g"`. (find path -type f will produce a list of all files if that is required!)
perl -pi -e 's/FINDTEXT/REPLACETEXT/' file* if FINDTEXT is one line.
perl -p0777i -e 's/FINDTEXT/REPLACETEXT/m' file* if FINDTEXT is many lines

29.3.3 vi

29.3.3.1 Modes

vi is used in one of many modes: command, find, edit/ insert, replace and view.
To shift between view and edit mode: press e or Escape. TO go to replace mode, press r.
To shift between view and command modes, press : or Escape. To shift between view and find mode, press / and Escape.

29.3.3.2 Commands and pattern matching

:g/Pattern/command executes command on every line matching a pattern.
`:1,$s/pattern/replacement/g`
text replacement with regular expressions in vi: use $\hat{C}\hat{M}$ for new line.

29.4 Resource check

cpuinfo, cat /proc/meminfo.
du -h –max-depth=1 gets disk usage.
du -hs gets total space used by directory files.

29.5 Image processing

Gimp, the gnome image viewer and editor.

29.6 Installation

Add a repository using the commandline or synaptic UI. Import necessary keys. update repository lists. Then install anything.

30 BASH scripting

30.1 Characteristic features

All environmental variables are imported into the context.

30.1.1 Command construction, execution

Very simple syntax for execution of commands: simply use a line which says: commandName. Construction of command-strings using variable names is also simple: someText $varName otherText

30.1.2 Logic limitations

Not suitable for logic more involved than an if else statement.

30.2 Writing and executing code

Bash commands and programs can either be run by providing/ typing the program in STDIN, or it can be run from a file.

30.2.1 Context

Every sequence of Bash commands is executed in a certain context. Different contexts do not share the variables (including environment variables like PWD and PATH) - so changing variables in one context does not affect another.

30.2.1.1 Current context

source fileName executes the file in the current context.

30.2.1.2 New context

One can run a file using: bash fileName.

Invocation as a command One can execute the file merely by saying fileName at the command prompt, if it begins with: #!/usr/bin/bash . This line is used by many interpreters to identify and use the interpreter appropriate to the file when one runs the file.
If #!/usr/bin/bash -x is used, all executed lines are echoed.

30.2.2 Environment variables

Some variables, called the environment variables, are set automatically when a shell context is created.
These variables are important because they are used while searching for various purposes affecting terminal display and command interpretation/ execution. Eg: `LD_LIBRARY_PATH`, PATH (a : separated list of directories where an executable file is to be sought), PWD (present working directory).

30.2.2.1 Setting

export VAR=value sets an environment variable.
To set these at startup, edit .bashrc and `.bash_aliases` in the home directory.
env lists the environment variables.

30.3 Variables and data

Dynamic typing.

30.3.1 Assignment

VARNAME=value. Array: area2=(zero one two three four) A particular element is set with area2[0]=val
Note that there should not be any space around =.

30.3.2 Reference

`$VAR_NAME ${VAR_NAME}` are references to the variable.
Array Element reference: `${area2[0]}`.
Reference the entire array: `${colors[@]}`.

31 Android development

Pocket computer usage is considered in another chapter.

31.1 Java SDK

Android development is done in Java. One can install the android SDK and google API following instructions from google. This also provides platform-tools for command-line access to the device.
This also creates a keystore for use while debugging (perhaps in signing google-api use agreements.)

31.1.1 C++ compiler

Native code can be used through JNI - C code will have to be compiled using ndk. Official NDK does not allow exceptions in code. Instead one can use "crystax NDK".

31.1.2 Logging

All prints to stdout and stderr done in the Java code is redirected to a common formatted log file. By default similar output of native code is sent to /dev/null. This can be changed getting root access and creating a file /data/local.prop with the following content:
log.redirect-stdio=true

31.1.3 Command line access to device

The command adb is useful. Parameters push and pull move files to and from the device. Parameter shell would connect to a very limited version of the linux environment available on the device.
[**Incomplete**]

31.2 Eclipse-plugin

Android development happens in Java, and there are nifty plugins for debugging the application using Eclipse. These tools are provide a layer over the command-line platform-tools provided by the Android SDK.

31.2.1 Debugging

Debugging can be launched from the java perspective and stopped from the Devices view in the DDMS perspective.
In the LogCat viewer in the DDMS perspective one can see various logger messages and lines printed out to stdout and stderr. The Console viewer in the same perspective tells us when the application has been built and sent to the device.
It includes a file explorer.
[**Incomplete**]

31.3 Application architecture

See the relavant note online.

31.3.1 Components

An app consists of components. Their types are Activities (UI), Services (background), ContentProviders (shared data management), BroadcastReceivers.

31.3.1.1 Communication with components

One expresses an intent to get a certain result from a component using Activity.startActivityForResult(intent, reqCode).
When the external component returns the result together with the intent object and a resultCode, Activity.onActivityResult is called.

31.3.2 Security and isolation

Following the principle of allowing minimal access, the applications must declare what resources they need in advance in AndroidManifest.xml. Every application is signed.
Every application runs in its own virtual machine (dalvik-VM).
[**Incomplete**]

31.3.3 Threading

Threads have MessageQueue-s, which are handled by Handler.handleMessage.

31.3.4 Resources

Resources required by the application are located in 'res' - including sound, image and xml files.

31.3.4.1 UI design

The look/ placement of UI components for various activity windows are recorded in xml files in the layout subfolder, which can be edited graphically using the interface provided by the Eclipse plugin.
Their functionality is defined in corresponding Activity classes. Java objects and classes which can be accessed and manipulated from code (which may be within the activity class) are generated automatically from the XML files.

Part VI

Distributed computing

32 Parallel computing paradigms

Many programs are 'embarassingly parallel' - so easy to parallelize.
There are 3 steps in any parallel algorithm: specification of the problems which must be solved in parallel, executing the problems in parallel, combining the results of these parallel executions.

33 Condor

This is a common job-scheduler. Once you specify the job in a certain file, condor tries to execute it on some processor (in a cluster) and retrieve the result.

33.1 Priority and restarts

Commonly jobs dispatched by condor are low priority - so if a higher priority process comes in, the condor job is stopped and moved to a different processor. If the program has checkpointing facilites, the job simply continues from where it stopped : Eg: Compiled languages. But if checkpointing is not available, the process restarts when moved to a different node: Eg: most interpreted languages.

34 ORC

34.1 Distinctive features of the language

34.1.1 Purpose of design

Inspired by functional programming languages.
Distributed computing: There'll be many services on the internet, need a language to orchestrate them. So, good internet mashup language.
Good for concisely reasoning about distrubted systems.

34.1.2 Sites

Everything is a site: a possibly remote function without side-effect which may not respond. Even +, -, if(..) etc.. are sites. These may return sites too. Actual site call is not executed until all arguments are available.

34.1.3 Functions

The language allows/ needs functions, but they're not site calls. It is simply a parametrized expression.

34.1.4 Parallelism

Highly parrellelizable.
Even a+b: a and b are evaluated independently.

34.1.5 Speed

34.2 Writing, Building and executing code

Run in browser, or using an eclipse plugin.

34.2.1 Debugging

34.3 Help

The website.

34.4 Pre-compilation processing

34.5 Variable and data types

34.5.1 Data types

signal is a unary data-type. boolean, numeric etc.. are other data types.
[] is an empty list.
a = Ref() yields a pointer: then do a.read(), a.write().

34.5.2 Declaration syntax

val valName = expression.
This boils down to pruning operator. Eg: $a|valName$ is actually $a|b < b < expression$.
It is scoped to the expression in the next line.
Pattern matching: $a : b = [1, 2, 3]; (\_, a) = (2, 3)$.

34.6 Data manipulation

34.6.1 Operators

/= is 'not equal to' boolean operator.

34.6.2 Accessing object methods

channel.get() actually is shorthand for calling a function which retrieves the get() function and then invokes it.

34.7 Code structure

34.7.1 Sentence syntax

There is actually just one sentence/ expression. Rest is syntactic sugar. Line breaks don't matter.

34.7.1.1 Commenting

{- asdf -}

34.7.2 Combinators

Use to stitch together expressions.
Parallel combinator: $a|b$.
Sequential/ push combinator: >. Eg: $prompt() > b > c(b)$. Syntactic sugar: $prompt() > b > c$ is same as $prompt() >> c$.

Pruning/ pull operator: $a < b$. Both sides begin to be evaluated simultaneously; if LHS needs a value, then it blocks.

34.7.3 Syntax of functions

Can be nested.
def fnName(arglist) = expression.
Function definition which uses pattern matching, usual syntactic sugar to specify termination condition:
fn([]) = asdfasdf
fn((a:asdf)) = asdfasdf

34.7.4 Decision, timing sites

if(). stop() never returns any value. RTimer().
Iteration accomplished through recursion.

34.7.5 Function invocation

fn(arguments) or fn.
fn(expression): syntactic sugar for $f(x) < x < expression$.

34.8 Error handling

34.9 Other library functions

34.9.1 User interaction I/O sites

prompt(), print().

Part VII

Serving web-pages

35 Common Gateway Interface (CGI)

35.1 HTTP server and dynamic content

A http server serves files; of which html files are normally viewed using web-browsers. These files may be static, or they may be generated dynamically, at the time of the request. For dynamic generation of such files, the http server should coordinate with some external program. At the minimum, it passes the request made by the client to this program and retrieves its output.

These programs may be precompiled - they may be written in C; or they may be written in a scripting language, and the http server may use an appropriate interpreter when the request is made.

35.2 Scripting mixed with html

The dynamic programs often dynamically generate html pages, so the dynamic portions of such pages can potentially be written in another language, while the static portions are written in html. Depending on the language used for the dynamic portion, this mixed-language is called by different names, like perlscript or JSP (or java server pages) or ASP.
The http server may utilize these scripts/ server pages by first converting it into a program of the corresponding language and then using the appropriate interpreter. Eg: JSP's are often first converted to Java servelets.

36 Client side scripting

Most browsers support javascript.

37 Yahoo pipes

With this service one can 'rewire the internet': that is, process data available on the web as we like.

37.1 Interface

Programming the processing pipeline is done with a simple graphical interface; where one connects boxes (representing various processing modules) with pipes (representing flow of information).

37.2 Looping

Processing modules may be of the following types: source modules which gather initial data published on the internet in various forms (html, rss feeds etc..), user input modules, various operator modules for modifications (splitting data, looping, filtering, string replacement, reformatting etc..).

38 Web API

Some web-services expose API to allow algorithmic access to the information they serve.

38.1 Request/ response format

They specify the serialization formats in which input/ request and output/ response objects (structured data) are to be encoded. This is usually in JSON or XML format.
XML, being more verbose, results in slower computation and transmission.

38.2 Underlying Protocol

Usually, they use http or rpc protocol.

38.3 Web interface Protocols

38.3.1 Simple Object access protocol (SOAP)

It specifies the use of XML for representing objects, certain Message Exchange Patterns (MEP).

39 Particular Web Api's

39.1 Wikimedia

The mediawiki api is specified here. Also, the bot-info page has more information.

39.1.1 Java wrappers

- jwpl
  Not useful in querying live data.
- JavaWikiBotFramework (jwbf)
  Seems to have good reviews, poor web documentation.
- gwtwiki bliki engine.
  Focuses mainly on converting between wiki text, plain text, google code wikitext and html etc..
  Has useful wiki (2) page with examples.
- java wiki api
  Has some examples.

Wiktionary frameworks:

- jwktl
  Not useful in querying live data.

40 Flash

Flash player security settings can be adjusted by visiting a special webpage.

Part VIII

Programming within browsers

41 Web Browsers

Many popular browsers enable one to synchronize the following across different computers: bookmarks, stored user information and passwords.
google-chrome is a fast browser - it benefits from Google's heavy efforts at making browsing fast. One can configure keywords to map to search engines - this makes searching easy. It has problems rendering hyperlinked indic text -esp dEvanAgarI- properly.

41.1 Extensions/ plugins

Users tend to write nifty plugins for their browsers. These are listed elsewhere.

41.2 User scripting

41.2.1 Interoperability

firefox, chrome, and IE all support user-scripting. The trend became popular with Greasemonkey scripts in firefox. Firefox user-scripts are managed using the greasemonkey extension, while similar extensions are available for chrome - but I haven't been able to make them work.
There are libraries which take browser differences into account.

41.3 Enhancements by websites

Very useful shortcuts are available with google mail and reader. They can be listed by pressing ?.

41.4 Observe browser activity

While programming, debugging or bug reporting, it is useful to observe details of what is happening internally. Eg: What web requests are being made; what javascript errors are observed.

In chrome: right click -¿ inspect element. The network and scripts tabs are very useful.

42 ECMA script/ javascript

42.1 Distinctive features

prototype-based object oriented. Functional (has first class functions, closures). Variable type is loosely set, and can dynamically change. C-like syntax.

42.1.1 Design flaws

Some design flaws lead to very confusing semantics, which makes bugs common both in writing and altering code.
"JavaScript has its share of design errors, such as the overloading of + to mean both addition and concatenation with type coercion, and the error-prone with statement should be avoided. The reserved word policies are much too strict."
Other problems: It takes varName = value; to mean a declaration of a global variable if varName had not been declared previously. DOM, which is heavily used by the language, is also flawed.
This, along with books promoting bad programming practices, use by amateur programmers, has led to it acquiring some bad reputation.

42.1.2 Programming discipline

JS's design flaws may be mitigated by adapting greater programming discipline. There are two steps: 1] Ensure syntactic correctness using jslint, 2] Ensure semantic correctness using proofs and testing as usual.

42.1.2.1 Development tools

Use built-in debuggers or extensions like firebug.
Use jslint to ensure syntactic quality in the code: jslint devel: true, browser: true, undef: false, vars: true, white: true, maxerr: 50, indent: 4
IDE: Bluefish.

42.1.3 Implementation variations

JavaScript and DOM implementations vary across browsers - even from the standards. So, scripts must be tested on different target browsers.

42.1.4 Acknowledgement

Much of the below is copied verbatim from the internet.

42.1.5 Security

“First, scripts run in a sandbox in which they can only perform web-related actions, not general-purpose programming tasks like creating files. Second, scripts are constrained by the same origin policy: scripts from one web site do not have access to information such as usernames, passwords, or cookies sent to another site.”
Security bugs are breaches of either the same origin policy (in cross-site scripting, or XSS) or the sandbox.

42.1.6 Document object model (DOM)

42.1.6.1 Structure

The document is viewed as a tree of objects/ node objects. Node objects, depending on their state (esp nodeType) and behavior can be : document nodes, text, attribute, element and other types of nodes.
At the root is the document node. The element nodes may correspond to either text nods or the various tags which appear in the html/ xml document.
See javascriptkit for a good list.

42.1.6.2 Document object

createElement(tagName) creates an element which may then be appended in the DOM tree.

42.1.6.3 Element object

For traversal: getElementsByTagName(’a’), getElementsById, childNodes[], nextSibling.
Accessing and altering attributes and text: attributes[], getAttribute, setAttribute, innerHTML.
Modifiers: appendChild.

42.2 UI

42.2.1 Events

Mouse events include mousedown, mouseup, mouseout, mousemove. These include pageX, pageY properties.

42.2.2 Event listeners

Event listeners may be added or removed to various node objects using addEventListener(eventTypeStr, listener, bCapture) and removeEventListener().

42.2.3 Getting selected text

Use document.getSelection().

42.3 Threads, timing

Threads are only now being added (2010).

42.3.1 Executing code after some time

timerId = setTimeout(functionHandle, milliseconds). This can be canceled with clearTimeout(timerId).

42.3.2 Periodic execution of code

setInterval(): syntax is similar to setTimeout. To cancel it, use clearInterval().

43 User scripts

43.1 Motivation

Modern browsers, perhaps aided by extensions like greasemonkey, allow users to write scripts which are activated for certain websites. The language is essentially javascript.
Many scripts can be found in userscripts.org.

43.2 Header format

```
// ==UserScript==
// @name           sanskrit vocabulary trainer
// @description    sanskrit vocabulary trainer
// @author         vishvAs vAsuki Iyengar
// @include        http://*vasuki*
// @version        1.0
// ==/UserScript==
```

Part IX

Databases

44 Database management systems

44.1 Relational Database management system (RDBMS)

All records can be viewed as consisting of some fields or attributes. Every attribute has a fixed domain. There are relations among these fields which impose constraints on what forms a valid record.
Tuples are a collection of values assigned to attributes. A bunch of tuples sharing the same attributes form a relation (or a table).

44.1.1 Relational algebra

Relations can undergo the following operations: Union, intersection, difference, cross product, selection/ restriction, projection, inner join, relational division (?).

44.1.2 Database normalization

What relations should exist and what attributes should various relations contain so that they do not redundantly store data?
To do this there are various normal forms (Boyce Codd). However, normalizing a database too much causes retrieval and update operations to slow down as they will involve joins.

44.2 Sharding

Databases can be too large to store in a single machine. In that case, the tables may be split horizontally and stored in different machines.
The different shards can be named using the convention tblName shardNum.
Some query engines like dremel can query multiple shards simultaneously.

44.3 Pooling writes

Database writes, when pooled, reduce client to database server traffic. Intelligent pools collect writes and flush when a sufficient load is accumulated.

45 Structured query language (SQL)

A standard language for dealing with RDBMS. So, data can be visualized as being stored in tables.

45.1 Help and tutorials

w3schools has a good tutorial.

45.2 Values

Strings are enclosed as here: 'asdf'.

45.3 Query language

45.3.1 Query table

The query table may be specified in several ways. In the most basic case, it is a single table.

table1 (as t1), table2 represents a cross-product, aka cross join. This returns a table whose rows are members of tbl1 X tbl2. In specifying temporary table names, as in table1 as t1, some dialects allow omission of 'as'.

tableList may be of the form: tbl1 join-command tbl2 [on condition]

Or it can be: (selectStatement)

45.3.2 Post selection processing

'group by fieldName' groups the data together before displaying.

Multiple select statements may be combined using keywords like: UNION ALL, INTERSECT.

45.3.3 Temporary tables

Temporary or inline tables are automatically created whenever a query table involves a join of any sort, or when it is defined by some select statement.
However, one may want to explicitly name and store the results of a query. Syntax is same as in create statement, with additional keyword: create temporary/ inline table.

45.3.3.1 Storage

These tables are stored in the RAM of the database client or on the database server.
The tables are dropped automatically at the end of the session.

45.4 Stored procedures

A piece of sql code which can be easily called.

46 Flat file DB

The entire database may be stored within a single file. Such databases may be useful when a database server is not required and when access is local. Eg: sqlite, berkeley db (not a relational db), recutils.

46.1 sqlite

sqlite3 databases cannot be opened with sqlite.
sqlite offers a limited set of field types.
sqlite does not allow joins during updates - making the process of copying columns a bigger chore.

46.1.1 GUI

sqlitebrowser offers a good GUI for sqlite - but the import from csv feature is very slow as of 2012.

46.1.2 Shell

46.1.2.1 Importing

```
.mode csv
.import fileName.csv tableName
```

47 MySql

47.1 Server

The database is stored in a file. Access to the database is mediated by a server. Different users can have different abilities. A root user is created at the time of installation.
By default the following databases/ schemata exist:

- mysql : Having tables such as user ..
- information_schema.

47.2 Clients

47.2.1 mysql shell

Within the shell, all commands not beginning with \ ends with a ;.

47.2.1.1 Invocation

Format: mysql [options] [database].
Important options include:
-u username -p.

47.2.1.2 General commands

`\q` quits.
General commands comply with the sql standard.

47.2.1.3 Meta-commands

show databases/ tables
'use database' selects a particular database to work with.

47.2.1.4 Administration

```
SET PASSWORD FOR 'root'@'localhost'= PASSWORD('secret_password');p

GRANT priv_type [(column_list)] [,priv_type [(column_list)] ...]
ON {tbl_name | * | *.* | db_name.*}
TO user_name [IDENTIFIED BY 'password']
[,user_name [IDENTIFIED BY 'password'] ...]
[WITH GRANT OPTION]
```

Correspondingly there is the REVOKE command.

47.2.2 Ancilliary commandline tools

mysqladmin provides easy access to some commands also available in mysql.
mysqldump creates database backups. mysqlhotcopy does the same, except it locks the databse while doing its job.
[**Incomplete**]

47.2.3 GUI

mysql administrator provides a useful interface to manage the server itself, the users, backups etc..
mysql querytool provides easy command reference and a window to easily enter commands and see results.
eclipse sql explorer [**Incomplete**].

47.3 Past experience

Provides unfriendly error messages, rejects sql commands accepted by sqlite.

Part X

Remote procedure call and serialization

Part XI

Document typesetting

48 Declarative vs imperative features

Many document display specification formats provide are declarative. On one hand, they specify the logical components of a document. On the other hand, they enable specification of display rules or styles for each logical element of the document.
Yet, despite this, sometimes finer imperative control over document display is desirable.
Some languages provide both features, while others provide one or the other.

49 latex

Both declarative (rule-based) and imperative specification of document display is possible.
A document element is defined to either be a command and or an environment - which are distinguished based on the input they accept.

49.1 Common commands, environments

Common commands include section, subsection, paragraph, subsubsection, title, author. Common environments include list (itemize and enumerate), tabular etc..

49.1.1 Templates

Rules which map commands and environments to imperative display rules are collected together in templates. These rules may be overriden. Every document must be based on some template.
Common templates include: report, article.

49.1.2 Packages

Often display rules may be redefined in files called packages, which may be used/ invoked using usepackage command.

49.1.2.1 Page setup

microtype for full justification of text. fullpage for using the maximum possible printable area.

49.1.2.2 List compaction

enumitem and shortlst make lists more configurable.

49.2 Bibliography

This part of the document is often dynamically generated based on citations used and a bibliography file. In this process, aux and bbl files are generated and used. ***In case of error, try deleting these files!***
Two things affect document display here: citation style and bibliographystyle. These are usually specified using packages, possibly followed by minor redefinitions of commands.

49.2.1 Packages

natbib is configurable and popular.

49.3 (Re)definition

Commands can be defined using: `\newcommand{cmd}[args]{def}`.
Environments can be defined using `\newenvironment{env}[args][default]{begdef}{enddef}`.
Older definitions can be overridden using renewcommand and renewenvironment.

49.4 Imperative display commands

textit, textbf.
center, raggedleft, raggedright.

49.4.1 Font size

In order: tiny scriptsize footnotesize small normalsize large Large LARGE huge Huge .
Using them resets the font size till the end of the current text block.

49.4.2 Display parameters

They can be set using addtolength and setlength commands.
Common parameters include textwidth, oddsidemargin, evensidemargin, topsidemargin, parskip.
One can define colors using: `\definecolor{mygrey}{gray}{.95}`.

49.4.3 Space management

`\vspace{-3ex}` sets the vertical spacing to a third the size of 'x'. This should ideally be used towards the end of the document.
hfill and vfill are used to fill spaces.

49.5 Drawing figures

pstricks. tikz: good for drawing graphs with nodes and edges, but not as advanced as pstricks.

50 html