

Clojure Cheat Sheet (Clojure 1.3, sheet v1.4)

Documentation

clojure.repl/ doc find-doc apropos source pst javadoc
(foo.bar/ is namespace for later syms)

Primitives

Numbers

Literals Long: 7 BigInt: 7N Ratio: -22/7 Double: 2.78
BigDecimal: 4.2M
Arithmetic + - * / quot rem mod inc dec max min
Compare = == not= < > <= >= compare
Bitwise bit-{and, or, xor, not, flip, set, shift-right,
shift-left, and-not, clear, test}
Cast byte short int long float double bigdec bigint
num rationalize biginteger
Test nil? identical? zero? pos? neg? even? odd?
Random rand rand-int
BigInt with-precision
Unchecked unchecked-{add, dec, divide, inc, multiply,
negate, remainder, subtract}-int

Strings

Create str format See also IO/to string
Use count get subs compare (clojure.string/) join escape
split split-lines replace replace-first reverse
(String) .indexOf .lastIndexOf
Regex #"pattern" re-find re-seq re-matches re-pattern
re-matcher re-groups (clojure.string/) replace
replace-first
Letters (clojure.string/) capitalize lower-case upper-case
Trim (clojure.string/) trim trim-newline triml trimr
Test char char? string? (clojure.string/) blank?

Other

Characters char char-name-string char-escape-string
Keywords keyword keyword? find-keyword
Symbols symbol symbol? gensym

Collections

Collections

Generic ops count empty not-empty into conj
Content tests distinct? empty? every? not-every? some
not-any?
Capabilities sequential? associative? sorted? counted?
reversible?
Type tests coll? list? vector? set? map? seq?

Lists

Create '() list list*
Examine first nth peek .indexOf .lastIndexOf
'Change' cons conj rest pop

Vectors

Create [] vector vec vector-of
Examine (my-vec idx) → (nth my-vec idx) get peek
.indexOf .lastIndexOf
'Change' assoc pop subvec replace conj rseq

Sets

Create #{ } set hash-set sorted-set sorted-set-by
Examine (my-set item) → (get my-set item) contains?
'Change' conj disj
Rel algebra (clojure.set/) join select project union
difference intersection
Get map (clojure.set/) index rename-keys rename map-invert
Test (clojure.set/) subset? superset?

Maps

Create {} hash-map array-map zipmap sorted-map
sorted-map-by bean frequencies group-by
Examine (:key my-map) → (get my-map :key) get-in
contains? find keys vals
'Change' assoc assoc-in dissoc merge merge-with
select-keys update-in
Entry key val
Sorted maps rseq subseq rsubseq

Transients (clojure.org/transients)

Create transient persistent!
Change conj! pop! assoc! dissoc! disj! Note: always use re-
turn value for later changes, never original!

Misc

Compare = == identical? not= not compare
clojure.data/diff
Test true? false? nil? instance?

Sequences

Creating a Lazy Seq

From collection seq vals keys rseq subseq rsubseq
From producer fn lazy-seq repeatedly iterate
From constant repeat range
From other file-seq line-seq resultset-seq
re-seq tree-seq xml-seq iterator-seq
enumeration-seq
From seq keep keep-indexed

Seq in, Seq out

Get shorter distinct filter remove for
Get longer cons conj concat lazy-cat mapcat cycle
interleave interpose
Tail-items rest nthrest fnext nnext drop drop-while
take-last for
Head-items take take-nth take-while butlast drop-last
for
'Change' conj concat distinct flatten group-by
partition partition-all partition-by split-at
split-with filter remove replace shuffle
Rearrange reverse sort sort-by compare
Process items map pmap map-indexed mapcat for replace seque

Using a Seq

Extract item first second last rest next ffirst nfirst
fnext nnext nth nthnext rand-nth when-first
max-key min-key
Construct coll zipmap into reduce reductions set vec
into-array to-array-2d
Pass to fn apply
Search some filter
Force evaluation doseq dorun doall
Check for forced realized?

Zippers (clojure.zip/)

Create zipper seq-zip vector-zip xml-zip
Get loc up down left right leftmost rightmost
Get seq lefts rights path children
'Change' make-node replace edit insert-child insert-left
insert-right append-child remove
Move next prev
Misc root node branch? end?

IO

to/from spit slurp (to writer/from reader, Socket, string with
... file name, URI, etc.)
to *out* pr prn print printf println newline (clo-
jure.pprint/) print-table
to writer (clojure.pprint/) pprint cl-format also: (binding
[*out* writer] ...)
to string format with-out-str pr-str prn-str print-str
println-str
from *in* read-line read
from reader line-seq read also: (binding [*in* reader] ...)
java.io.Reader
from string read-string with-in-str
Open with-open (clojure.java.io/) text: reader writer
binary: input-stream output-stream
Binary (.write ostream byte-arr) (.read
istream byte-arr) java.io.OutputStream
java.io.InputStream GitHub: gloss byte-spec
Misc flush (.close s) file-seq *in* *out* *err*
(clojure.java.io/) file copy GitHub: fs

Functions

Create	fn defn defn- definline identity constantly memfn comp complement partial juxt memoize fn! every-pred some-fn
Call	-> ->> apply
Test	fn? ifn?

Abstractions

Protocols (clojure.org/protocols)

Define	(defprotocol Slicey (slice [at]))
Extend	(extend-type String Slicey (slice [at] ...))
Extend null	(extend-type nil Slicey (slice [_] nil))
Reify	(reify Slicey (slice [at] ...))

Records (clojure.org/datatypes)

Define	(defrecord Pair [h t])
Access	(:h (Pair. 1 2)) → 1
Create	Pair. ->Pair map->Pair

Types (clojure.org/datatypes)

Define	(deftype Pair [h t])
Access	(.h (Pair. 1 2)) → 1
Create	Pair. ->Pair
	(deftype Pair [h t]
With methods	Object (toString [this] (str "<" h ", " t ">")))

Multimethods (clojure.org/multimethods)

Define	(defmulti my-mm dispatch-fn)
Method define	(defmethod my-mm :dispatch-value [args] ...)
Dispatch	get-method methods
Remove	remove-method remove-all-methods
Prefer	prefer-method prefers
Relation	derive isa? parents ancestors descendants make-hierarchy

Macros

Create	defmacro definline macroexpand-1 macroexpand
Branch	and or when when-not when-let when-first if-not if-let cond condp case
Loop	for doseq dotimes while
Arrange	.. doto ->
Scope	binding locking time with-{in-str, local-vars, open, out-str, precision, redefs, redefs-fn}
Lazy	lazy-cat lazy-seq delay
Doc.	assert comment doc

Reader Macros

'	Quote 'form → (quote form)
\	Character literal
;	Single line comment
^	Metadata (see Metadata section)
@	Deref @form → (deref form)
'	Syntax-quote
~	Unquote
~@	Unquote-splicing
#"p"	Regex Pattern p
#'	Var quote #'x → (var x)
#()	#(...) → (fn [args] (...))
#_	Ignore next form

Metadata (clojure.org/special_forms)

General	^{:key1 val1 :key2 val2 ...}
Abbrevs	^Type → ^{:tag Type}, ^:key → ^{:key true}
Common	^:dynamic ^:private ^:static ^:const
Examples	(defn ^:private ^:static ^String my-fn ...) (def ^:dynamic *dyn-var* val)
On Vars	meta with-meta vary-meta alter-meta! reset-meta! doc find-doc test

Special Forms (clojure.org/special_forms)

def	if do let quote var fn loop recur throw try monitor-enter monitor-exit
Binding Forms /	(examples) let fn defn defmacro loop for
Destructuring	doseq if-let when-let

Vars and global environment (clojure.org/vars)

Def variants	def defn defn- definline defmacro defmethod defmulti defonce defrecord
Interned vars	declare intern binding find-var var
Var objects	with-local-vars var-get var-set alter-var-root var?
Var validators	set-validator! get-validator

Namespace

Current	*ns*
Create/Switch	(tutorial) ns in-ns create-ns
Add	alias def import intern refer
Find	all-ns find-ns
Examine	ns-{name, aliases, map, interns, publics, refers, imports}
From symbol	resolve ns-resolve namespace
Remove	ns-unalias ns-unmap remove-ns

Loading

Load libs	(tutorial) require use import refer
List loaded	loaded-libs
Load misc	load load-file load-reader load-string

Concurrency

Atoms	atom swap! reset! compare-and-set!
Futures	future future-{call, done?, cancel, cancelled?} future?
Threads	bound-fn bound-fn* {get, push, pop}-thread-bindings thread-bound?
Misc	locking pcalls pvalues pmap seque promise deliver

Refs and Transactions (clojure.org/refs)

Create	ref
Examine	deref @ (@form → (deref form))
Transaction	sync dosync io!
In transaction	ensure ref-set alter commute
Validators	set-validator! get-validator
History	ref-history-count ref-{min, max}-history

Agents and Asynchronous Actions (clojure.org/agents)

Create	agent
Examine	agent-error
Change state	send send-off restart-agent
Block waiting	await await-for
Ref validators	set-validator! get-validator
Watchers	add-watch remove-watch
Thread handling	shutdown-agents
Error	error-handler set-error-handler! error-mode set-error-mode!
Misc	*agent* release-pending-sends

Java Interoperation (clojure.org/java_interop)

General	.. doto Classname/ Classname. new bean comparator enumeration-seq import iterator-seq memfn set!
Cast	boolean byte short char int long float double bigdec bigint num cast biginteger
Exceptions	throw try catch finally pst

Arrays

Create	make-array {object, boolean, byte, short, char, int, long, float, double}-array aclone to-array to-array-2d into-array
Use	aget aset aset-{boolean, byte, short, char, int, long, float, double} alength amap areduce
Cast	booleans bytes shorts chars ints longs floats doubles

Proxy

Create	proxy get-proxy-class {construct, init}-proxy
Misc	proxy-mappings proxy-super update-proxy

Other

XML	clojure.xml/parse xml-seq
REPL	*1 *2 *3 *e *print-dup* *print-length* *print-level* *print-meta* *print-readably*
Code	*compile-files* *compile-path* *file* *warn-on-reflection* compile gen-class gen-interface loaded-libs test
Misc	eval force hash name *clojure-version* clojure-version *command-line-args*