Clojure Cheat Sheet (Clojure 1.7 - 1.10, sheet v52)

Documentation

clojure.repl/ doc find-doc apropos dir source pst javadoc (foo.bar/ is

namespace for later syms)

Primitives

Numbers Literals

Long: 7, hex Oxff, oct 017, base 2 2r1011, base 36 36rCRAZY BigInt: 7N Ratio: -22/7 Double: 2.78 -1.2e-5 BigDecimal:

4.2M

+ - * / quot rem mod inc dec max min +' -' *' inc' dec' Arithmetic

Compare == < > <= >= compare

Ritwise bit-and bit-or bit-xor bit-not bit-flip bit-set

bit-shift-right bit-shift-left bit-and-not bit-clear bit-test unsigned-bit-shift-right (see BigInteger for integers larger

than Long)

Cast byte short int long float double bigdec bigint num rationalize

biginteger

zero? pos? neg? even? odd? number? rational? integer? ratio? decimal? float? (1.9) double? int? nat-int? Test

neg-int? pos-int?
rand rand-int

Random BigDecimal with-precision

Unchecked *unchecked-math* unchecked-add unchecked-dec unchecked-inc

unchecked-multiply unchecked-negate unchecked-subtract

Strings

Create str format "a string" "escapes \b\f\n\t\r\" octal \377 hex \ucafe"

See also section IO/to string

Use

count get subs compare (clojure.string/) join escape split split-lines replace replace-first reverse (1.8) index-of last-index-of

#"pattern" re-find re-seq re-matches re-pattern re-matcher re-groups
(clojure.string/) replace replace-first re-quote-replacement Note: \ Regex

in #"" is not escape char. (re-pattern "\\s*\\d+") can be written #"\s*\d+"

Letters

(clojure.string/) capitalize lower-case upper-case Trim

(clojure.string/) trim trim-newline trim1 trimr string? (clojure.string/) blank? (1.8) starts-with? ends-with? Test

includes?

Other char char? char-name-string char-escape-string literals: \a Characters

\newline (more at link)

Keywords keyword keyword? find-keyword literals: :kw :my.name.space/kw

::in-cur-namespace ::namespace-alias/kw symbol symbol? gensym literals: my-sym my.ns/foo literals: true false nil Symbols

Misc

Collections

Collections

Create

count empty not-empty into conj (clojure.walk/) walk prewalk Generic ops

prewalk-demo prewalk-replace postwalk postwalk-demo

postwalk-replace (1.9) bounded-count Content tests

distinct? empty? every? not-every? some not-any? sequential? associative? sorted? counted? reversible? coll? list? vector? set? map? seq? record? (1.8) Capabilities Type tests

map-entry?

Lists (conj, pop, & peek at beginning)

Create () list list*

first nth peek .indexOf .lastIndexOf cons conj rest pop Examine

'Change

Vectors (conj, pop, & peek at end)

[] vector vec vector-of mapv filterv (clojure.core.rrb-vector/) vector

vec vector-of

(my-vec idx) \rightarrow (nth my-vec idx) get peek .indexOf .lastIndexOf 'Change assoc assoc-in pop subvec replace conj rseq update update-in

Ops reduce-kv

Sets Create unsorted #{} set hash-set

sorted-set sorted-set-by (clojure.data.avl/) sorted-set sorted-set-by (flatland.ordered.set/) ordered-set (clojure.data.int-Create sorted

map/) int-set dense-int-set

 $(my\text{-set item}) \rightarrow (get my\text{-set item}) contains?$ Change' conj disj (clojure.set/) union difference intersection select See also sec-

Set ops

tion Relations (clojure.set/) subset? superset?

Test

Sorted sets rseq subseq rsubseq

Examine

Examine

Create unsorted {} hash-map array-map zipmap bean frequencies group-by (clo-

jure.set/) index

sorted-map sorted-map-by (clojure.data.avl/) sorted-map Create sorted

sorted-map-by (flatland.ordered.map/) ordered-map (clojure.data.priority-map/) priority-map (flatland.useful.map/)

ordering-map (clojure.data.int-map/) int-map

(my-map k) \rightarrow (get my-map k) also (:key my-map) \rightarrow (get my-map :key) get-in contains? find keys vals

'Change assoc assoc-in dissoc merge merge-with select-keys update update-in (clojure.set/) rename-keys map-invert GitHub: Medley

reduce-kv Ops Entry kev val

rseq subseq rsubseq Sorted maps

Queues (coni at end. peek & pop from beginning)

Create clojure.lang.PersistentQueue/EMPTY (no literal syntax or constructor fn)

Examine 'Change conj pop

Relations (set of maps, each with same keys, aka rels)

Rel algebra $({\it clojure.set}/) \ {\it join select project union difference intersection}$

index rename

Transients (clojure.org/reference/transients)

transient persistent! Create

conj! pop! assoc! dissoc! disj! Note: always use return value for Change

later changes, never original!

Misc

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

Test

Sequences

Creating a Lazy Seq

From collection eq vals keys rseq subseq rsubseq sequence

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 sea keep keep-indexed

Seq in, Seq out

Get shorter distinct filter remove take-nth for dedupe random-sample Get longer cons conj concat lazy-cat mapcat cycle interleave interpose Tail-items rest nthrest next fnext nnext drop drop-while take-last for Head-items take take-while butlast drop-last for

conj concat distinct flatten group-by partition partition-all partition-by split-at split-with filter remove replace shuffle $\,$

reverse sort sort-by compare
map pmap map-indexed mapcat for replace seque Process items

Rearrange Using a Seq

Check for forced

'Change'

Extract item first second last rest next ffirst nfirst fnext nnext nth

nthnext rand-nth when-first max-key min-key

zipmap into reduce reductions set vec into-array to-array-2d mapv filterv Construct coll

apply some filter Pass to fn Search Force evaluation doseq dorun doall run!

realized? Transducers (clojure.org/reference/transducers)

Off the shelf map mapcat filter remove take take-while take-nth drop

drop-while replace partition-by partition-all keep keep-indexed map-indexed distinct interpose cat dedupe

random-sample (1.9) halt-when

Create your own completing ensure-reduced unreduced See also section Concur-

rency/Volatiles

Use into sequence transduce eduction

Early termination reduced reduced? deref

Spec (rationale, guide)

valid? conform unform explain explain-data explain-str Operations

explain-out form describe assert check-asserts check-asserts?

gen exercise exercise-fn

Generator ops Defn. & registry def fdef registry get-spec spec? spec with-gen

Logical and or Collection coll-of map-of every every-kv keys merge

Regex cat alt * + ? & keys*

int-in inst-in double-in int-in-range? inst-in-range? Range nilable multi-spec fspec conformer
explain-printer *explain-out* Other Custom explain

Predicates with test.check generators

number? rational? integer? ratio? decimal? float? zero? (1.9) double? int? nat-int? neg-int? pos-int? Numbers (1.9) double? int? nat-int? neg-int? pos-int? keyword? symbol? (1.9) ident? qualified-ident? qualified-keyword? qualified-symbol? simple-ident? Symbols keywords

simple-keyword? simple-symbol? string? true? false? nil? some? (1.9) boolean? bytes? Other scalars inst? uri? uuid?

list? map? set? vector? associative? coll? sequential? seq? empty? (1.9) indexed? seqable? Collections

(1.9) any? Other Ю

to writer

to/from spit slurp (to writer/from reader, Socket, string with file name, URI, etc.) to *out* pr prn print printf println newline (clojure.pprint/) print-table

(clojure.pprint/) pprint cl-format also: (binding [*out* writer]

format with-out-str pr-str prn-str print-str println-str to string from *in* read-line (clojure.edn/) read (clojure.tools.reader.edn/) read line-seq (clojure.edn/) read (clojure.tools.reader.edn/) read also: from reader

thereseq (cojure.com/) read cojure.com/, juva.io.Reader with-in-str (clojure.edn/) read-string (clojure.tools.reader.edn/) from string

read-string

Open with-open (clojure.java.io/) text: reader writer binary: input-stream output-stream (.write ostream byte-arr) (.read istream byte-arr) Binary

java.io.OutputStream java.io.InputStream GitHub: gloss byte-spec

Misc flush (.close s) file-seq *in* *out* *err* (clojure.java.io/) file copy delete-file resource as-file as-url as-relative-path $\mathsf{GitHub}\colon \ \mathsf{fs}$

Data readers *data-readers* default-data-readers *default-data-reader-fn* (1.10) tap> add-tap remove-tap tap

Functions

fn defn defn- definline identity constantly memfn comp complement Create

partial juxt memoize fnil every-pred some-fn

Call apply -> ->> trampoline as-> cond-> cond->> some->> fn? ifn?

Test

```
Abstractions (Clojure type selection flowchart)
                                                                                               Special Forms (clojure.org/reference/special_forms)
Protocols (clojure.org/reference/protocols)
                                                                                                  def if do let letfn quote var fn loop recur set! throw try monitor-enter
 Define
                ( defprotocol Slicey (slice [at]))
 Extend
                  extend-type String Slicey (slice [at] ...))
                                                                                                  Binding Forms
                                                                                                                     (examples) let fn defn defmacro loop for doseq if-let
 Extend null
                ( extend-type nil Slicey (slice [_] nil))
                                                                                                 Destructuring
                                                                                                                     when-let if-some when-some
  Reify
                ( reify Slicey (slice [at] ...))
  Test
                satisfies? extends?
                                                                                               Vars and global environment (clojure.org/reference/vars)
  Other
                extend extend-protocol extenders
                                                                                                 Def variants
                                                                                                                  def defn defn- definline defmacro defmethod defmulti defonce
Records (clojure.org/reference/datatypes)
                                                                                                                  defrecord
                                                                                                  Interned vars
                                                                                                                  declare intern binding find-var var
            ( defrecord Pair [h t])
 Define
                                                                                                  Var objects
                                                                                                                  with-local-vars var-get var-set alter-var-root var? bound?
 Access
           (:h (Pair. 12)) \rightarrow 1
                                                                                                                   thread-bound?
           Pair. ->Pair map->Pair record?
  Create
                                                                                                  Var validators
                                                                                                                  set-validator! get-validator
 Test
Types (clojure.org/reference/datatypes)
                                                                                               Namespace
 Define
                   ( deftype Pair [h t])
                                                                                                  Current
                                                                                                                    *ns*
                   (.h (Pair. 1 2)) → 1
Pair. ->Pair
  Access
                                                                                                  Create/Switch
                                                                                                                   (tutorial) ns in-ns create-ns
 Create
                                                                                                  Add
                                                                                                                   alias def import intern refer
                   ( deftype Pair [h t]
                                                                                                  Find
                                                                                                                   all-ns find-ns
                    Object (toString [this] (str "<" h "," t ">")))
 With methods
                                                                                                  Examine
                                                                                                                   ns-name ns-aliases ns-map ns-interns ns-publics ns-refers
                                                                                                                   ns-imports
                                                                                                  From symbol
                                                                                                                   resolve ns-resolve namespace the-ns (1.10) requiring-resolve
Multimethods (clojure.org/reference/multimethods)
                                                                                                 Remove
                                                                                                                   ns-unalias ns-unmap remove-ns
 Define
                   ( defmulti my-mm dispatch-fn)
  Method define
                   ( defmethod my-mm :dispatch-value [args] ...)
                                                                                               Loading
                   get-method methods
  Dispatch
                                                                                                 Load libs
                                                                                                                (tutorial) require use import refer
 Remove
                   remove-method remove-all-methods
                   prefer-method prefers
  Prefer
                                                                                                 Load misc
                                                                                                                load load-file load-reader load-string
 Relation
                   derive underive isa? parents ancestors descendants
                   make-hierarchy
                                                                                                             atom swap! reset! compare-and-set! (1.9) swap-vals! reset-vals!
                                                                                                  Atoms
Datafy (article)
                                                                                                  Futures
                                                                                                              future future-call future-done? future-cancel future-cancelled?
 Datafy
           (clojure.datafy/) datafy nav
                                                                                                              future?
                                                                                                  Threads
                                                                                                              bound-fn bound-fn* get-thread-bindings push-thread-bindings
                                                                                                             pop-thread-bindings thread-bound? volatile! vreset! vswap! volatile?
Macros
                                                                                                  Volatiles
                                                                                                              locking pcalls pvalues pmap seque promise deliver
                                                                                                  Misc
 Create
             defmacro definline
  Debug
             macroexpand-1 macroexpand (clojure.walk/) macroexpand-all
                                                                                               Refs and Transactions (cloiure.org/reference/refs)
 Branch
             and or when when-not when-let when-first if-not if-let cond condp
                                                                                                  Create
                                                                                                                  ref
             case when-some if-some
                                                                                                                  \texttt{deref @ (@form} \rightarrow (\texttt{deref form}))
                                                                                                  Examine
             for doseq dotimes while
 Loop
                                                                                                  Transaction
                                                                                                                  sync dosync io!
 Arrange
                 doto -> ->> as-> cond-> cond->> some->>
 Scope
             binding locking time with-in-str with-local-vars with-open
                                                                                                  In transaction
                                                                                                                  ensure ref-set alter commute
                                                                                                  Validators
                                                                                                                  set-validator! get-validator
             with-out-str with-precision with-redefs with-redefs-fn
                                                                                                  History
                                                                                                                  ref-history-count ref-min-history ref-max-history
             lazy-cat lazy-seq delay
  Lazy
 Doc
             assert comment doc
                                                                                               Agents and Asynchronous Actions (clojure.org/reference/agents)
                                                                                                  Create
                                                                                                                     agent
Special Characters (clojure.org/reference/reader, guide)
                                                                                                  Examine
                                                                                                                     agent-error
                                                                                                                     send send-off restart-agent send-via
                                                                                                  Change state
                       Comma reads as white space. Often used between map key/value
                       pairs for readability. quote: 'form \rightarrow ( quote form)
                                                                                                                     set-agent-send-executor! set-agent-send-off-executor!
                                                                                                                     await await-for
set-validator! get-validator
                                                                                                  Block waiting
                                                                                                  Ref validators
                        Namespace separator (see Primitives/Other section)
                                                                                                  Watchers
                                                                                                                     add-watch remove-watch
                       Character literal (see Primitives/Other section)
                                                                                                  Thread handling
                       Keyword (see Primitives/Other section)
                                                                                                                     shutdown-agents
                                                                                                                     error-handler set-error-handler! error-mode set-error-mode!
                       Single line comment
                                                                                                  Misc
                                                                                                                     *agent* release-pending-sends
                       Metadata (see Metadata section)
  *foo*
                        'earmuffs' - convention to indicate dynamic vars, compiler
                       warns if not dynamic
                                                                                               Java Interoperation (clojure.org/reference/java_interop)
                       \texttt{Deref: @form} \xrightarrow{\bullet} (\texttt{deref form})
                                                                                                                .. doto Classname/ Classname. new bean comparator
                       Syntax-quote
                                                                                                                enumeration-seq import iterator-seq memfn set! class class?
                        'auto-gensym', consistently replaced with same
 foo#
                                                                                                                bases supers type gen-class gen-interface definterface
                       auto-generated symbol everywhere inside same '( ... )
                                                                                                 Cast
                                                                                                                boolean byte short char int long float double bigdec bigint num
                       Unquote
                                                                                                                cast biginteger
 ~@
                       Unquote-splicing
                                                                                                                throw try catch finally pst ex-info ex-data Throwable->map (1.9)
                                                                                                 Exceptions
                        'thread first' macro ->
                                                                                                               StackTraceElement->vec (1.10) ex-cause ex-message (clojure.main/)
                        'thread last' macro ->>
  ->>
                                                                                                               ex-triage ex-str err->msg report-error
       <!! >! <!
                        core.async channel macros >!! <!! >!
                                                                                               Arrays
                       List literal (see Collections/Lists section)
                        Vector literal (see Collections/Vectors section)
                                                                                                 Create
                                                                                                           make-array object-array boolean-array byte-array short-array
                       Map literal (see Collections/Maps section) Var-quote #'x \rightarrow ( var x)
                                                                                                            char-array int-array long-array float-array double-array aclone to-array to-array-2d into-array
 #"
                       \#"p" reads as regex pattern p (see Strings/Regex section)
                                                                                                            aget aset aset-boolean aset-byte aset-short aset-char aset-int
                                                                                                  Use
                       Set literal (see Collections/Sets section)
 #{
                                                                                                            aset-long aset-float aset-double alength amap areduce
                       Anonymous function literal: #(...) \rightarrow (fn [args] (...))
Anonymous function argument: %N is value of anonymous function arg N. % short for %1. %k for rest args.
                                                                                                            booleans bytes shorts chars ints longs floats doubles
 %
                                                                                               Proxy (Clojure type selection flowchart)
                       Reader conditional: #?(:clj x :cljs y) reads as x on JVM,
                                                                                                           proxy get-proxy-class construct-proxy init-proxy
  #?
                                                                                                  Create
                       y in ClojureScript, nothing elsewhere. Other keys: :cljr:default
                                                                                                  Misc
                                                                                                            proxy-mappings proxy-super update-proxy
 #70
                       Splicing reader conditional: [1 #?@(:clj [x y] :cljs
                                                                                               Zippers (clojure.zip/)
                       [w z]) 3] reads as [1 x y 3] on JVM, [1 w z 3] in
                                                                                                  Create
                                                                                                              zipper seq-zip vector-zip xml-zip
                       ClojureScript, [1 3] elsewhere.
                                                                                                  Get loc
                                                                                                              up down left right leftmost rightmost
                       tagged literal e.g. #inst #uuid map namespace syntax e.g. #:foo{:a 1 :b 2} is equal to
 #foo
                                                                                                              lefts rights path children
                                                                                                  'Change
                                                                                                              make-node replace edit insert-child insert-left insert-right
                        {:foo/a 1 :foo/b 2}
                                                                                                              append-child remove
                       (1.9) symbolic values: ##Inf ##-Inf ##NaN
 ##
                                                                                                  Move
                                                                                                              next prev
                        JavaContainerClass$InnerClass
                                                                                                 Misc
                                                                                                             root node branch? end?
 foo?
                       conventional ending for a predicate, e.g.: zero? vector?
                       instance? (unenforced)
                                                                                               Other
                       conventional ending for an unsafe operation, e.g.: set! swap! alter-meta! (unenforced)
 fool
                                                                                                  XMI
                                                                                                             clojure.xml/parse xml-seq
                                                                                                  REPL
                                                                                                             *1 *2 *3 *e *print-dup* *print-length* *print-level* *print-meta*
                        conventional name for an unused value (unenforced)
                                                                                                             *print-readably*
 #
                       Ignore next form
                                                                                                             *compile-files* *compile-path* *file* *warn-on-reflection* compile
                                                                                                 Code
                                                                                                             loaded-libs test
                                                                                                             eval force hash name *clojure-version* clojure-version
Metadata (clojure.org/reference/reader, special_forms)
                                                                                                  Misc
                                                                                                              *command-line-args*
              ^{:key1 val1 :key2 val2 ...} 
^Type \rightarrow ^{:tag Type}, ^:key \rightarrow ^{:key true} 
^:dynamic ^:private ^:doc ^:const
 General
                                                                                                             (clojure.java.browse/) browse-url (clojure.java.shell/) sh with-sh-dir
                                                                                                  Browsei
  Abbrevs
                                                                                                  / Shell
                                                                                                             with-sh-env
  Common
                                                         (def ^:dynamic *dyn-var*
 Examples
              (defn ^:private ^String my-fn ...)
              val)
 On Vars
              meta with-meta vary-meta alter-meta! reset-meta! doc find-doc
```