Documentation

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

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

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

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

Cast byte short int long float double bigdec bigint num rationalize

biginteger

zero? pos? neg? even? odd? number? rational? integer? ratio? decimal? float?

Random rand rand-int

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

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

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

(clojure.string/) capitalize lower-case upper-case Letters (clojure.string/) trim trim-newline triml trimr Trim

Test string? (clojure.string/) blank? (1.8) starts-with? ends-with? includes?

Other

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

(more at link)

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

::in-cur-namespace ::namespace-alias/kw

symbol symbol? gensym literals: my-sym my.ns/foo literals: true false nil Symbols

Misc

Collections

Collections

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

distinct? empty? every? not-every? some not-any? Content tests sequential? associative? sorted? counted? reversible? Capabilities

coll? list? vector? set? map? seq? record? (1.8) map-entry? Type tests

Lists (conj. pop. & peek at beginning)

Create () list list*

first nth peek .indexOf .lastIndexOf Examine

Change cons conj rest pop

Vectors (conj, pop, & peek at end)

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

vector-of

Examine $(\texttt{my-vec idx}) \, \rightarrow \, (\, \, \texttt{nth my-vec idx}) \, \, \texttt{get peek .indexOf .lastIndexOf}$

'Change' assoc assoc-in pop subvec replace conj rseq update-in (1.7) update

Ops

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-

map/) int-set dense-int-set

Examine $(\text{my-set item}) \rightarrow (\text{get my-set item}) \text{ contains}?$

conj disj 'Change

(clojure.set/) union difference intersection select See also section Set ops

Relations

(clojure.set/) subset? superset? Sorted sets rseq subseq rsubseq

Maps

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 Examine

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

assoc assoc-in dissoc merge merge-with select-keys update-in 'Change (1.7) update (clojure.set/) rename-keys map-invert GitHub: Medley

Ops reduce-kv Entry key val

Sorted maps rseq subseq rsubseq

Queues (conj at end, peek & pop from beginning)

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

fn) peek Examine 'Change' conj pop Relations (set of maps, each with same keys, aka rels)

(clojure.set/) join select project union difference intersection index Rel algebra

Transients (clojure.org/reference/transients)

Create transient persistent!

conj! pop! assoc! dissoc! disj! Note: always use return value for later changes,

never original!

Misc

Change

= identical? not= not compare clojure.data/diff

true? false? instance? nil? some?

Sequences

Compare

Creating a Lazy Seq

From collection seq 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 seq keep keep-indexed

Seq in, Seq out

Get shorter distinct filter remove take-nth for (1.7) 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 'Change' conj concat distinct flatten group-by partition partition-all

partition-by split-at split-with filter remove replace shuffle reverse sort sort-by compare

Rearrange Process items map pmap map-indexed mapcat for replace seque

Using a Seg

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

nthnext rand-nth when-first max-key min-key zipmap into reduce reductions set vec into-array to-array-2d Construct coll

mapv filterv

Pass to fn apply some filter Search

Force evaluation doseq dorun doall (1.7) run!

Check for forced 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 (1.7) cat dedupe random-sample

(1.9) halt-when Create your own (1.7) completing ensure-reduced unreduced See also section Concur-

rency/Volatiles

into sequence (1.7) transduce eduction Use Early termination reduced reduced? deref

Spec (rationale, guide)

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

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

Generator ops gen exercise exercise-fn

Defn. & registry def fdef registry get-spec spec? spec with-gen Logical and or

coll-of map-of every every-kv keys merge Collection

cat alt * + ? & keys* Regex Range int-in inst-in double-in int-in-range? inst-in-range?

Other nilable multi-spec fspec conformer Custom explain explain-printer *explain-out*

Predicates with test.check generators

number? rational? integer? ratio? decimal? float? zero? (1.9)

double? int? nat-int? neg-int? pos-int? keyword? symbol? (1.9) ident? qualified-ident? qualified-keyword?

Symbols qualified-symbol? simple-ident? simple-keyword? simple-symbol? keywords Other string? true? false? nil? some? (1.9) boolean? bytes? inst? uri?

scalars uuid? Collections

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

(1.9) any? Other

IO

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

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 (clojure.tools.reader.edn/) read line-seq (clojure.tools.reader.edn/) read also: (binding [*in* reader] from reader

..) java.io.Reader

from string with-in-str (clojure.tools.reader.edn/) read-string 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 flush (.close s) file-seq *in* *out* *err* (clojure.java.io/) file copy delete-file resource as-file as-url as-relative-path GitHub:

Data readers *data-readers* default-data-readers *default-data-reader-fn*

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?

Protocols (clojure.org/reference/protocols) Def variants def defn defn- definline defmacro defmethod defmulti defonce Define (defprotocol Slicey (slice [at])) defrecord Interned vars declare intern binding find-var var extend-type String Slicey (slice [at] ...)) Extend Extend null Var objects with-local-vars var-get var-set alter-var-root var? bound? (extend-type nil Slicey (slice [_] nil)) Reify (reify Slicey (slice [at] ...)) thread-bound? set-validator! get-validator Test satisfies? extends? Other extend extend-protocol extenders Namespace Records (clojure.org/reference/datatypes) Current *ns* Define (defrecord Pair [h t]) Create/Switch (tutorial) ns in-ns create-ns Access (:h (Pair. 1 2)) \rightarrow 1 Add alias def import intern refer Create Pair. ->Pair map->Pair Find all-ns find-ns record? Test Examine ns-name ns-aliases ns-map ns-interns ns-publics ns-refers ns-imports Types (clojure.org/reference/datatypes) resolve ns-resolve namespace the-ns From symbol Define (deftype Pair [h t]) Remove ns-unalias ns-unmap remove-ns $(.h (Pair. 1 2)) \rightarrow 1$ Create Pair. ->Pair Loading (deftype Pair [h t] With methods Load libs (tutorial) require use import refer Object (toString [this] (str "<" h "," t ">"))) List loaded loaded-libs load load-file load-reader load-string Multimethods (clojure.org/reference/multimethods) Define (defmulti my-mm dispatch-fn) Concurrency Method define (defmethod my-mm :dispatch-value [args] ...) atom swap! reset! compare-and-set! (1.9) swap-vals! reset-vals! Atoms Dispatch get-method methods future future-call future-done? future-cancel future-cancelled? **Futures** Remove remove-method remove-all-methods future? Prefer prefer-method prefers Threads bound-fn bound-fn* get-thread-bindings push-thread-bindings Relation derive underive isa? parents ancestors descendants make-hierarchy pop-thread-bindings thread-bound? Volatiles (1.7) volatile! vreset! vswap! volatile? Macros Misc locking pcalls pvalues pmap seque promise deliver Create defmacro definline Refs and Transactions (clojure.org/reference/refs) Debug macroexpand-1 macroexpand (clojure.walk/) macroexpand-all Create ref and or when when-not when-let when-first if-not if-let cond condp case Branch $deref @ (@form \rightarrow (deref form))$ Examine when-some if-some Transaction sync dosync io! for doseq dotimes while Loop In transaction ensure ref-set alter commute . doto -> ->> as-> cond-> cond->> some-> some->> Arrange Validators set-validator! get-validator Scope binding locking time with-in-str with-local-vars with-open with-out-str History ref-history-count ref-min-history ref-max-history with-precision with-redefs with-redefs-fn Lazy lazy-cat lazy-seq delay Agents and Asynchronous Actions (clojure.org/reference/agents) assert comment doc Doc. Create agent agent-error Examine Special Characters (clojure.org/reference/reader, guide) Change state send send-off restart-agent send-via set-agent-send-executor! Comma reads as white space. Often used between map key/value pairs for set-agent-send-off-executor! readability. Block waiting await await-for Ref validators ightarrow (quote form) set-validator! get-validator Watchers Namespace separator (see Primitives/Other section) add-watch remove-watch Character literal (see Primitives/Other section) Thread handling shutdown-agents Error error-handler set-error-handler! error-mode set-error-mode! Keyword (see Primitives/Other section) Single line comment Misc *agent* release-pending-sends Metadata (see Metadata section) 'earmuffs' - convention to indicate dynamic vars, compiler warns *foo* Java Interoperation (clojure.org/reference/java_interop) if not dynamic .. doto Classname/ Classname. new bean comparator enumeration-seq 0 Deref: @form → (deref form) import iterator-seq memfn set! class class? bases supers type Syntax-quote gen-class gen-interface definterface boolean byte short char int long float double bigdec bigint num cast 'auto-gensym', consistently replaced with same auto-generated foo# symbol everywhere inside same '(...) Unquote Exceptions throw try catch finally pst ex-info ex-data (1.9) ~@ Unquote-splicing StackTraceElement->vec 'thread first' macro -> ->> 'thread last' macro ->> Arrays >!! <!! >! <! core.async channel macros >!! <!! >! <! make-array object-array boolean-array byte-array short-array char-array Create List literal (see Collections/Lists section) int-array long-array float-array double-array aclone to-array Vector literal (see Collections/Vectors section) to-array-2d into-array Map literal (see Collections/Maps section) aget aset aset-boolean aset-byte aset-short aset-char aset-int aset-long Use $Var-quote #'x \rightarrow (var x)$ aset-float aset-double alength amap areduce #"p" reads as regex pattern p (see Strings/Regex section) Cast booleans bytes shorts chars ints longs floats doubles Set literal (see Collections/Sets section) Anonymous function literal: $\#(...) \rightarrow (fn [args] (...))$ #1 Proxy (Clojure type selection flowchart) #(Anonymous function argument: %N is value of anonymous function % proxy get-proxy-class construct-proxy init-proxy Create arg N. % short for %1. %% for rest args. proxy-mappings proxy-super update-proxy #? (1.7) Reader conditional: #?(:clj x :cljs y) reads as x on JVM, y in ClojureScript, nothing elsewhere. Other keys: :cljr Zippers (clojure.zip/) :default Create zipper seq-zip vector-zip xml-zip #70 (1.7) Splicing reader conditional: [1 #?@(:clj [x y] :cljs [w Get loc up down left right leftmost rightmost z]) 3] reads as [1 x y 3] on JVM, [1 w z 3] in ClojureScript, [1 Get seq lefts rights path children 3] elsewhere. 'Change' make-node replace edit insert-child insert-left insert-right #foo tagged literal e.g. #inst #uuid append-child remove JavaContainerClass\$InnerClass Move next prev conventional ending for a predicate, e.g.: zero? vector? foo? Misc root node branch? end? instance? (unenforced) foo! conventional ending for an unsafe operation, e.g.: set! swap! Other alter-meta! (unenforced) XML clojure.xml/parse xml-seq conventional name for an unused value (unenforced) REPL *1 *2 *3 *e *print-dup* *print-length* *print-level* *print-meta* Ignore next form *print-readably* Code *compile-files* *compile-path* *file* *warn-on-reflection* compile Metadata (clojure.org/reference/reader, special_forms) loaded-libs test eval force hash name *clojure-version* clojure-version General ^{:key1 val1 :key2 val2 ...} Misc \rightarrow ^{:tag Type}, ^:key \rightarrow ^{:key true} *command-line-args* Abbrevs Type ^:dynamic ^:private ^:doc ^:const (defn ^:private ^String my-fn ...) (clojure.java.browse/) browse-url (clojure.java.shell/) sh with-sh-dir Browser Common (def ^:dynamic *dyn-var* val) Examples meta with-meta vary-meta alter-meta! reset-meta! doc find-doc test

Vars and global environment (clojure.org/reference/vars)

Abstractions (Clojure type selection flowchart)

Special Forms (clojure.org/reference/special_forms)

if-some when-some

monitor-exit Binding Forms /

Destructuring

def if do let letfn quote var fn loop recur set! throw try monitor-enter

(examples) let fn defn defmacro loop for doseq if-let when-let