

# Introduction to Clojure

Guide to baking for the intrepid robot



## Baking Commands

(grab :egg)	Pick something up.
(squeeze)	Squeeze whatever you are holding.
(release)	Release what you are holding.
(scoop :flour)	Fill cup in hand with an ingredient.
(add-to-bowl)	Add what you are holding to the bowl.
(mix)	Mix the ingredients in the bowl.
(pour-into-pan)	Pour the bowl into the pan.
(bake-pan 30)	Put the pan in the oven.
(cool-pan)	Put the pan on the cooling racks. Returns the id of the cooling rack.

Name of ingredient; you can also grab the :cup.

## Fetching commands

(go-to :pantry)	Change locations in the bakery.
(load-up :egg)	Pick up an ingredient as cargo.
(unload :egg)	Put down an ingredient from cargo.

Name of location to go to.

Name of ingredient.

## Filling orders

(get-morning-orders)	Get a list of orders.
(delivery receipt)	Give delivery bot a receipt for baked goods.

Receipt contains order id, cooling rack ids, and delivery address.

## General commands

(status)	Print out the status of the bakery.
(start-over)	Reset the state of the simulator.
(bakery-help)	Print out a list of commands.

## Ingredients

:egg	squeezed	:fridge
:butter	simple	:fridge
:milk	scooped	:fridge
:flour	scooped	:pantry
:cocoa	scooped	:pantry
:sugar	scooped	:pantry

## Locations

:fridge	:egg, :milk, :butter
:pantry	:flour, :cocoa, :sugar
:prep-area	where baking is done

## **Simple ingredients**

Simple ingredients are simply grabbed then added to the bowl.

(grab :butter)  
(add-to-bowl)

## **Squeezed ingredients**

Squeezed ingredients must be squeezed after grabbing them, then added to the bowl.

(grab :egg)  
(squeeze)  
(add-to-bowl)

## **Scooped ingredients**

Scooped ingredients use the cup. Grab the cup, scoop, then add to bowl. Be sure to release the cup after you're done.

(grab :cup)  
(scoop :flour)  
(add-to-bowl)  
(release)

## Cake recipe

2 cups flour  
2 eggs  
1 cup milk  
1 cup sugar

1. mix all ingredients
2. bake in pan for 25 minutes
3. let cool

## Cookie recipe

1 egg  
1 cup flour  
1 cup sugar  
1 stick butter

1. mix in bowl
2. bake in pan 30 minutes
3. let cool

## Brownie recipe

2 cups flour  
2 eggs  
1 cup sugar  
2 cups cocoa  
1 cup milk  
2 butters

1. mix together butter, sugar, and cocoa in a bowl
2. then add flour, eggs, and milk to same bowl and mix
3. bake in pan for 35 minutes
4. let cool

*TWO separate mixing steps!!*

## Orders

Orders come in in the morning. You can get the morning orders with (get-morning-orders) or (get-morning-orders-day3). They are given to you as a list. Each order looks like this:

```
{:orderid 123
  :address "323 Robot Ln"
  :items {:cake 14
           :cookies 12}}
```

Keep track of the order id  
Address must be passed onto the delivery bot.  
Baked good's name and quantity.

As a baker, you are responsible for managing the orders you receive. That means fetching the required ingredients, baking all of the goods, and giving a receipt to the delivery bot.

## Receipts

Receipts should be created for each order and given to the delivery bot for prompt delivery. Please ensure that you create one receipt for each order. It confuses the delivery bot to have multiple receipts per order.

```
{:orderid 123
  :address "323 Robot Ln"
  :rackids [:cooling-rack-324
             :cooling-rack-325
             :cooling-rack-326]}
```

Order id corresponds from order.  
Address comes from the order.  
A list of cooling rack ids where the delivery bot can find the baked goods.

# Clojure guide

## Defining a named function

```
(defn add-egg []
  (grab :egg)
  (squeeze)
  (add-to-bowl))
```

Function name  
Arguments  
Body  
Return value is value of last expression

The diagram illustrates the structure of a Clojure named function definition. It points to the first '(defn' as 'Function name', the parameters 'add-egg' and '[]' as 'Arguments', and the three expressions '(grab :egg)', '(squeeze)', and '(add-to-bowl)' as 'Body'. A final annotation states 'Return value is value of last expression' pointing to the closing ')'.

## Conditional

```
(cond
  condition => (= x 100)
  expression => (println "one hundred")
  condition => (> x 10)
  expression => (println "greater than ten")
  condition => :else <----- :else is always truthy
  expression => (println "something else"))
```

## Shorter conditional

```
(if (scooped? :flour)
  then expression => "makes sense"
  else expression => "that's weird")
```

condition

The diagram shows the structure of a Clojure if-expression. It points to the '(if' keyword as 'condition', the ':flour' argument as 'then expression', and the quoted string 'that's weird' as 'else expression'.

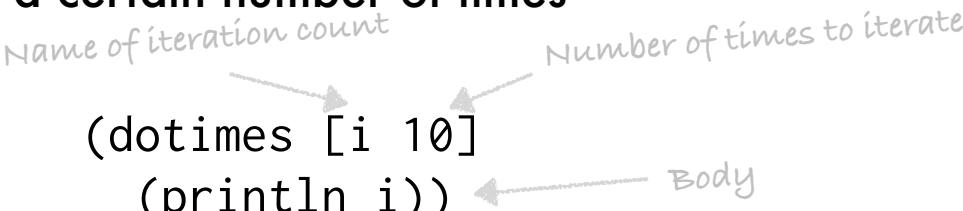
## Do things in sequence

```
(do
  (println "first step")
  (println "second step"))
```



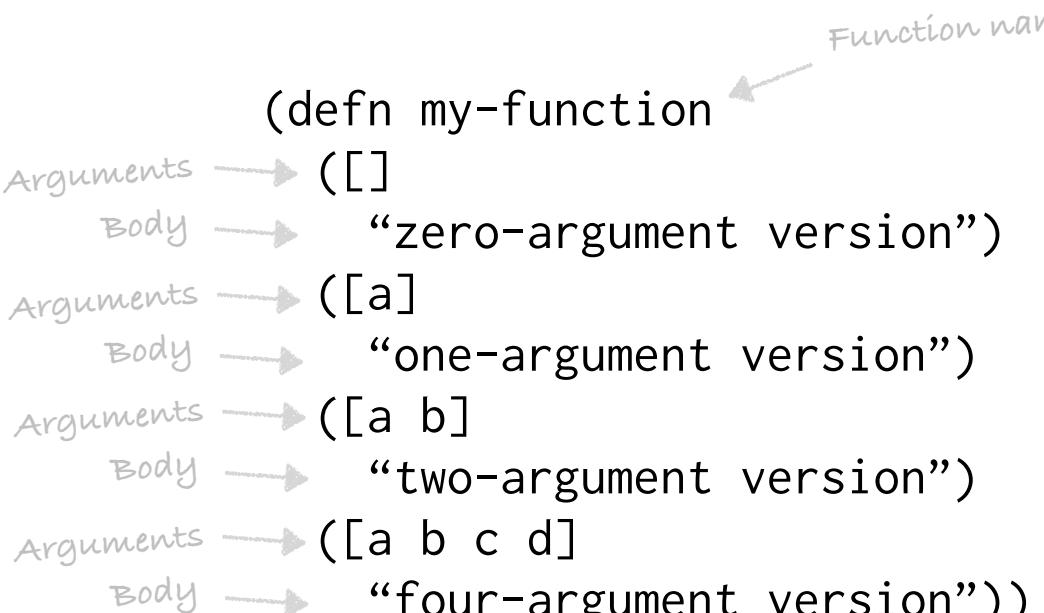
## Do things a certain number of times

```
(dotimes [i 10]
  (println i))
```



## One function, different number of arguments

```
(defn my-function
  [Arguments] Body)
  ([Arguments] Body)
  ([Arguments] Body)
  ([Arguments] Body))
```



## Rest argument (any number of arguments in a list)

```
(defn my-function [& rs]  
  (println rs))
```

Any number of  
arguments

## Apply a function to a list of arguments

```
(apply println args)
```

Function

List of arguments

## Sets

```
#{}  
Make empty set
```

## Does a set contain an element?

```
(contains? #{1 2 3 "Hello!"} "Hello!")
```

Set

Element

## Define a global variable

```
(def my-variable 1)
```

Name of variable

value

## Maps (hash maps)

```
{  
  Make an empty hash map
```

{}  
Make an empty hash map

## Get the value given a key

(get ingredients :flour)

Some map

Key to look up

## Check if a map contains a given key

(contains? ingredients :milk)

Some map

Key to look up

## An even shorter conditional, often used for effects

(when (scooped? :flour)  
  Body  
    → (println "Flour is scooped")  
    → :ok)

condition

## Iterate through a collection (or sequence)

(doseq [x [1 4 2 3 4]]  
  Body  
    → (println x))

Variable

Collection

## Define variables in local scope

(let [x 1  
     y 2  
     z (+ x y)]  
  Body  
    → (println z))

Name

Value

Name

Value

Name

Value

**Merge two maps (a and b) using a combining function**

(merge-with + a b)

**Add items to a collection**

(into {} [:x 1] [:x 2] [:x 5])

**Transform one list into another**

(for [x {:x 1 :y 2 :z 3}]  
 Body → (count x))

**Combine items in a collection into a single value**

(reduce + 0 [1 4 5 3 90 23 2 32])

**Create a list of 10 items, 0 through 9**

(range 10)

**Group elements into a hash map based on return values**

(group-by even? [1 2 3 4 5 6])

## Create an anonymous function

Arguments →  
(fn [x y] (\* x (+ 3 y)))  
Body

# Leiningen commands

Leiningen is the most popular Clojure project management tool. These commands should be run at the terminal (command prompt).

## Start a REPL

```
lein repl
```

## Run the -main function

```
lein run
```

## Get a list of Leiningen directives

```
lein help
```