Deckz

v0.3.1   2025-09-18   GPL-3.0-or-later

Render poker-style cards and full decks.

MICHELE DUSI

 @micheledusi

DECKZ is a flexible and customizable package to render and display poker-style playing cards in Typst[1]. Use it to visualize individual cards, create stylish examples in documents, or build full decks and hands for games and illustrations.

Table of Contents

---

[1] https://typst.app/

Part I
How to use DECKZ

DECKZ is a Typst package designed to **display playing cards in the classic poker style**, using the standard French suits (hearts ♥, diamonds ♦, clubs ♣, and spades ♠). Whether you need to show a single card, a hand, or a full deck, DECKZ provides flexible tools to visualize cards in a variety of formats and layouts. The package has been developed with the idea of it being used in games, teaching materials, or any document where clear and attractive card visuals are needed.

This manual for DECKZ is organized in three main parts:
1. Section I helps you get started with the **main features**;
2. Section II provides **detailed documentation** for each function, serving as a reference;
3. Section III presents **practical examples** that combine different features.

At the end, you'll find credits and instructions for contributing to the project.

> This manual refers to the most recent DECKZ release as of today: **version 0.3.1**.

I.1 Import the package

To start using DECKZ in your Typst document, simply **import the package** with:

```
#import "@preview/deckz:0.3.1"
```

This makes all DECKZ functions available under the `deckz` namespace. Congratulations, you're now ready to start visualizing cards!

I.2 Basic usage

The main entry point to start using DECKZ visualization features is the `#deckz.render` function. Here's an example of how to use it:

```
#deckz.render("7D", format: "large")
```
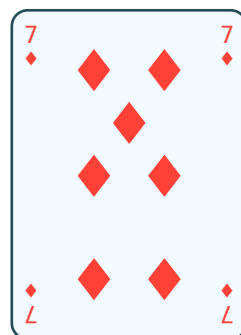

As you can see, the function has been called with two arguments, and it produced a large card with the rank 7 and the suit of diamonds ♦.

1. The first argument is the **card identifier** as a string (#deckz.render.card). Use **standard short notation** like "AH", "10S", "QC", etc., where the first letter(s) indicates the **rank**, and the last letter the **suit**.

   - **Available ranks**: A, 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K.
   - **Available suits**: H (Hearts ♥), D (Diamonds ♦), C (Clubs ♣), S (Spades ♠).

   > 🔭 *Behind The Scenes.* Card identifier is **case-insensitive**, so "as" and "AS" are equivalent and both represent the *Ace of Spades*.

2. The second argument is optional and specifies the **format** of the card display (#deckz.render.format). If not provided, DECKZ functions will typically default to medium, which is a well-balanced card size suitable for most uses.

   To learn more about the available formats, see Section I.2.1.

I.2.1 Formats

DECKZ provides multiple **display formats** to fit different design needs:

Format	Description
inline	A minimal format where the rank and suit are shown directly inline with text.
mini	The smallest visual format: a tiny rectangle with the rank on top and the suit at the bottom.
small	A compact but clear card with rank in opposite corners and the suit centered.
medium	A full, structured card with proper layout, two corner summaries, and realistic suit placement.
large	An expanded version of medium with corner summaries on all four sides for maximum readability.
square	A balanced 1:1 format with summaries in all corners and the main figure centered.

Here's an example of how the same card looks in different formats:

```
#deckz.render("5S", format: "inline")
#deckz.render("5S", format: "mini")
#deckz.render("5S", format: "small")
#deckz.render("5S", format: "medium")
#deckz.render("5S", format: "large")
#deckz.render("5S", format: "square")
```
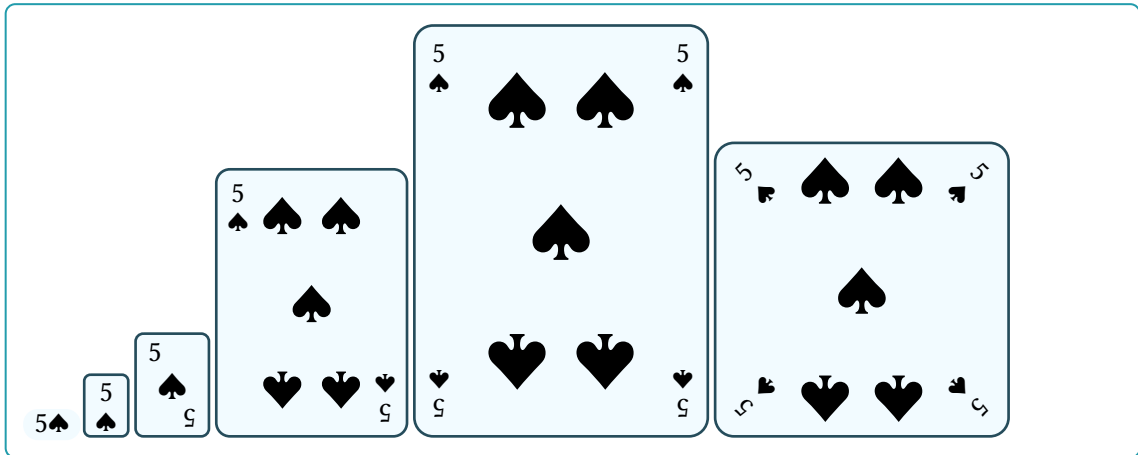
You can use any of these with the **general function** #deckz.render, or by calling directly the
specific format functions:

- #deckz.inline,
- #deckz.mini,
- #deckz.small,
- #deckz.medium,
- #deckz.large,
- #deckz.square.

> 🔭 *Behind The Scenes.* All formats are **responsive to the current text size**: they scale proportionally
> using em units, making them adaptable to different layouts and styles.
>
> For reference, the summary view contained in larger formats (i.e. the symbols representing the rank and
> suit of a cards, usually placed in the card's corners) **scale with the current text size**, ensuring that card
> details remain readable by being no smaller than 1em in size.

```
#deckz.mini("2C")
#deckz.medium("JH")
#deckz.square("AD")

are equivalent to

#deckz.render("2C", format: "mini")
#deckz.render("JH", format: "medium")
#deckz.render("AD", format: "square")
```
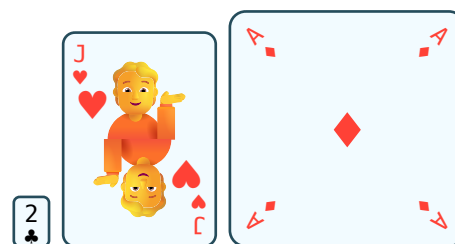

are equivalent to
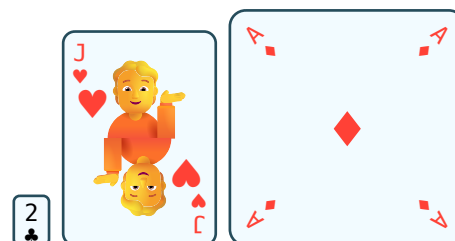
If you want more examples of how to use these formats, check out Section III at the end of this document.

I.2.2 Back of cards

To render the **back of a card**, you can use the `#deckz.back` function. This will display a generic card back design, which can be useful for games or when you want to hide the card's face.

```
This is the back of a card:
#deckz.back(format: "small")
```

This is the back of a card:

Alternatively, you can use the general `#deckz.render` function with `"back"` as the card code, which is equivalent.

> 🔭 *Behind The Scenes.* Any string other than a valid card identifier will be interpreted as a request for the back of the card (except for the empty string). The convention, however, is to use the string `"back"` for clarity.

```
// These are all equivalent:
#deckz.medium("back")
#deckz.render("back", format: "medium")
#deckz.back(format: "medium")
```
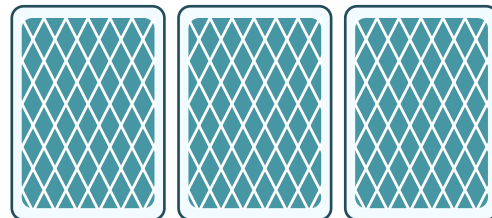
> 🚀 *Coming Soon Feature.* Currently, the back of cards uses a fixed design. In future updates, DECKZ will allow you to customize the back of cards and decks.

I.3 Visualize cards together

DECKZ also provides convenient functions to render **entire decks** or **hands of cards**. Both functions produce a *CeTZ* canvas from the CETZ[2] package, which can be used in any context where you need to display multiple cards together.

I.3.1 Decks

The deck visualization is created with the `#deckz.deck` function, which takes a card identifier as an argument. It renders a full deck of cards, with the specified card on top.

---

[2]https://typst.app/universe/package/cetz

```
#deckz.deck("6D")
```


In the `#deckz.deck` function, you can also specify different parameters to **customize deck appearance**; we list here some of them.

- `#deckz.deck.angle` – The direction towards which the cards are shifted.
- `#deckz.deck.height` – The height of the deck, represented as a `length`.
- `#deckz.deck.format` – The format of the cards in the deck. It can be any of the formats described above, such as `inline`, `mini`, `small`, `medium`, `large`, or `square`.

For more information and in-depth explanations, see the documentation in Section II.

```
#stack(
  dir: ltr,
  spacing: 1cm,
  deckz.deck("8S"),
  deckz.deck("8S", angle: 90deg, height: 2.5cm),
  deckz.deck("8S", angle: 180deg, height: 8pt, format: "small"),
  deckz.deck("8S", angle: 80deg, height: 18mm, noise: 0.1)
)
```
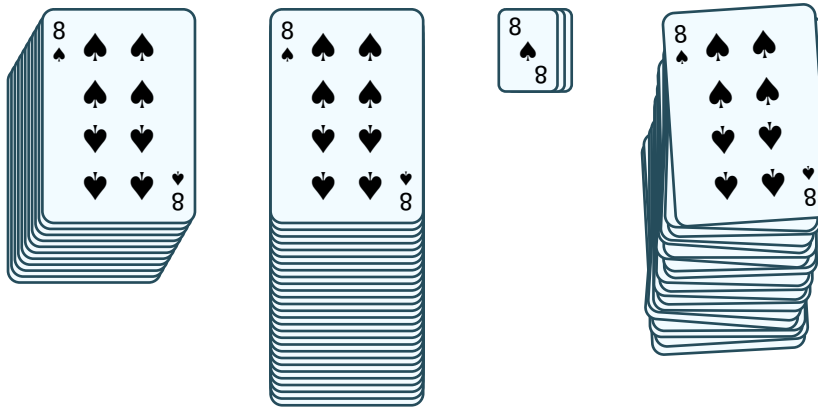

I.3.2 Line

The line visualization is created with the `#deckz.line` function, which takes a variable number of card identifiers as arguments. It renders an horizontal **line of cards**, with the specified cards displayed side by side. It's a simple way to show a sequence of cards in a row, minimizing the space they occupy.

```
#deckz.line("AC", "KH", "QS", "JD", "10S")
```
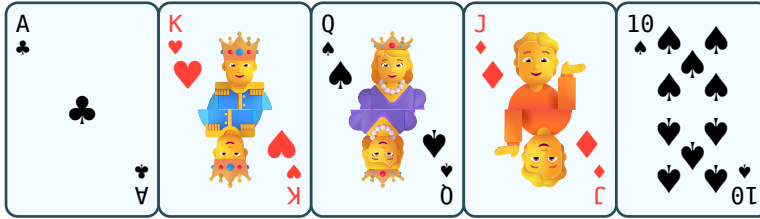

As can be seen in the example above, the cards are displayed adjacent with no extra space.

However, such display can be customized. You can use the following parameters (more parameters for `#deckz.line` explained in Section II):

- `#deckz.line.spacing` – The **spacing between adjacent cards**. It must be a `length` (e.g., `10pt`). If set to `auto`, visualization will place cards according to the specidied width (if any).
- `#deckz.line.width` – The **total width** of the line. It can be a `length` (e.g., `350pt`) or a `ratio` (e.g., `90%`). If left to `auto`, the width is determined by the number of cards and their format.

> 🔭 *Behind The Scenes.* Being one dependent on the other, only **one parameter** between `#deckz.line.spacing` and `#deckz.line.width` can be set to a specific value at a time. If none of them is set, cards are placed adjacent with no extra space.

- `#deckz.line.format` – The **format** of the cards in the line. It can be any of the formats described above, such as `"inline"`|`"mini"`|`"small"`|`"medium"`|`"large"`|`"square"`. The default is `"medium"`.

See also Section II for the full list of parameters of `#deckz.line`.

```
#let my-line = ("AS", "KH", "QD", "JS", "JH", "10C", "9D", "6C")

#deckz.line(..my-line, format: "mini")
#deckz.line(..my-line, format: "small", spacing: 10pt)
#deckz.line(..my-line, format: "mini", width: 6cm)
#deckz.line(..my-line, format: "small", spacing: 1cm, noise: 0.8)
#deckz.line(..my-line, format: "small", width: 100%)
#deckz.line(..my-line, format: "square", width: 90%)
```
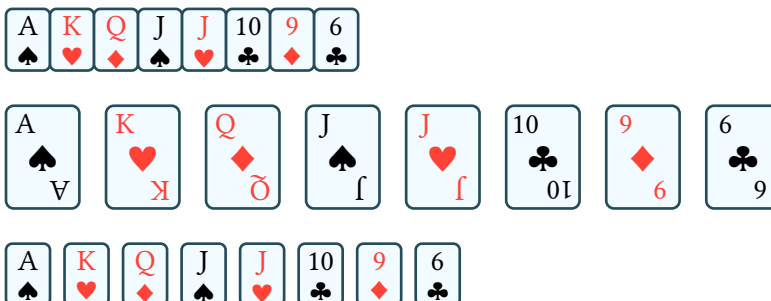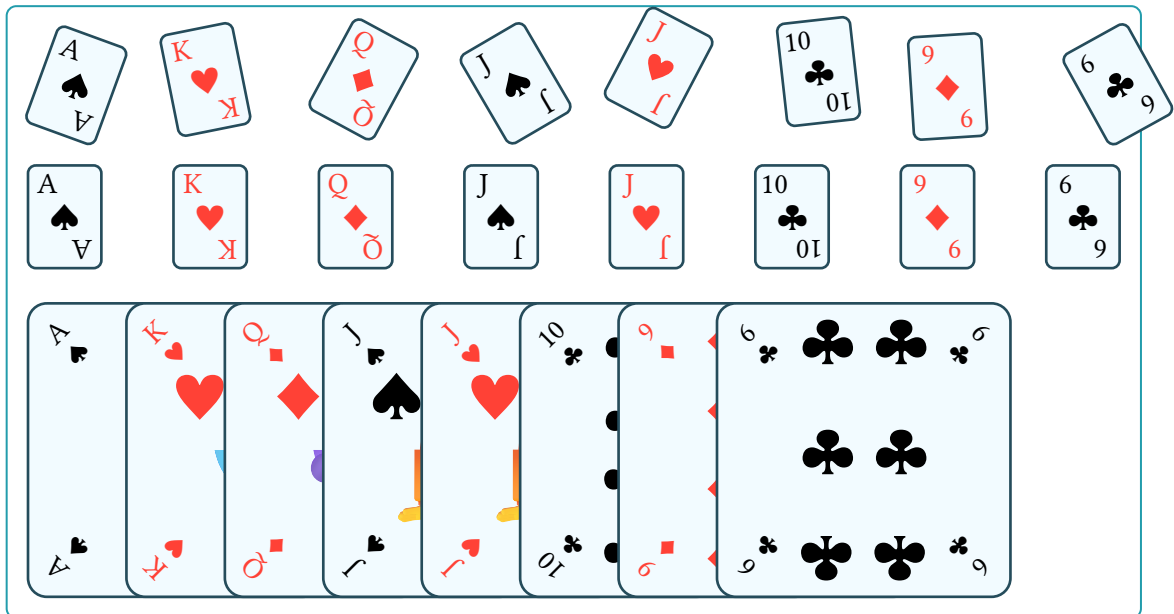
I.3.3 Hands

The hand visualization is created with the `#deckz.hand` function, which takes a variable number of card identifiers as arguments. It renders a **hand of cards**, with the specified cards displayed side by side.

```
#deckz.hand("AS", "KS", "QS", "JS", "10S")
```
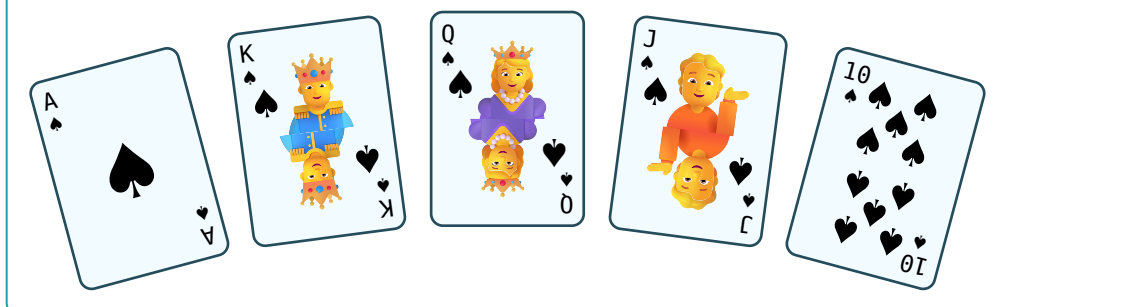

As can be seen in the example above, the cards are displayed in an arc shape, with the first card on the left and the last card on the right.

To customize such display, you can use the following parameters (more parameters for `#deckz.hand` explained in Section II):

- `#deckz.hand.angle` – The angle of the arc in degrees, i.e. the angle between the first and last cards' orientations.
- `#deckz.hand.width` – The width of the hand, i.e. the distance between the centers of the first and last card.
- `#deckz.hand.format` – The format of the cards in the deck. It can be any of the formats described above, such as `inline`, `mini`, `small`, `medium`, `large`, or `square`. The default is `medium`.

```
#let my-hand = ("AS", "KH", "QD", "JS", "JH", "10C", "9D", "6C")

#table(
  columns: (1fr),
  align: center,
  stroke: none,
  deckz.hand(..my-hand),
  deckz.hand(angle: 0deg, width: 4cm, ..my-hand),
  deckz.hand(format: "mini", ..my-hand),
  deckz.hand(width: 5cm, noise: 0.8, format: "small", ..my-hand),
  deckz.hand(angle: 180deg, width: 3cm, noise: 0.1, format: "large", ..(my-
hand + my-hand)),
)
```
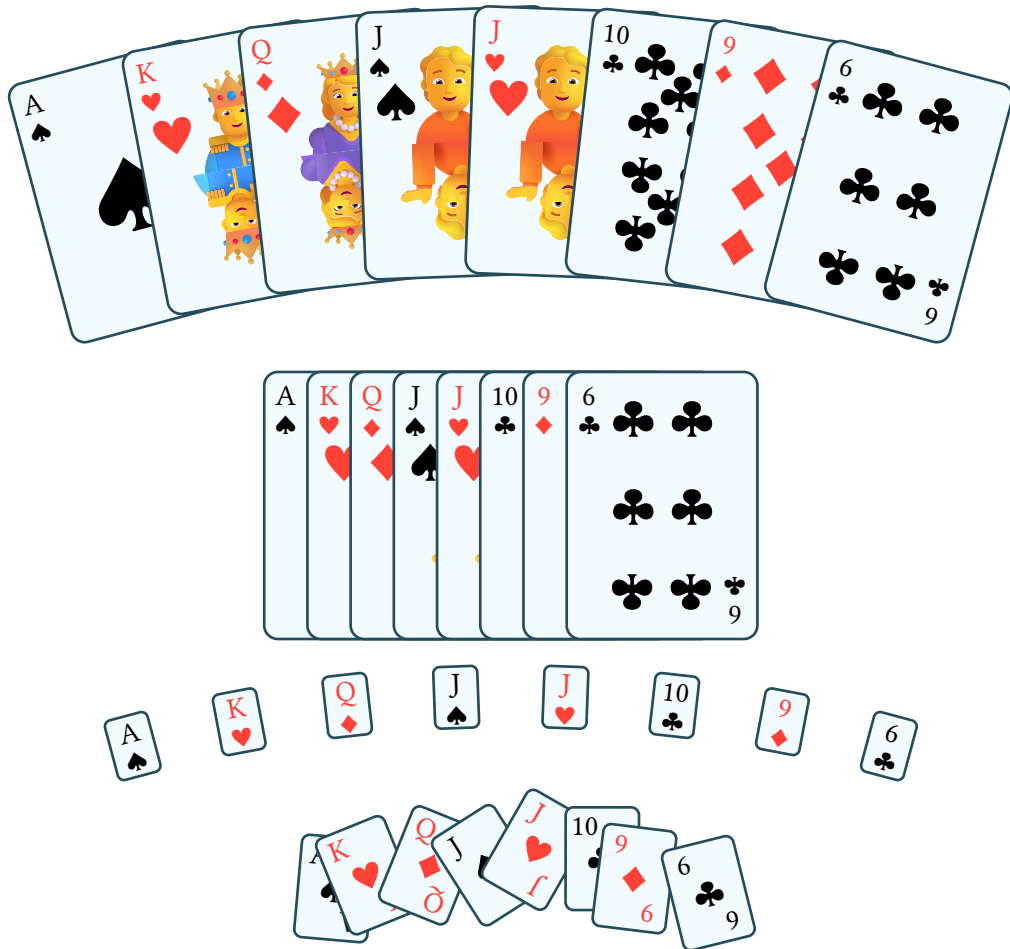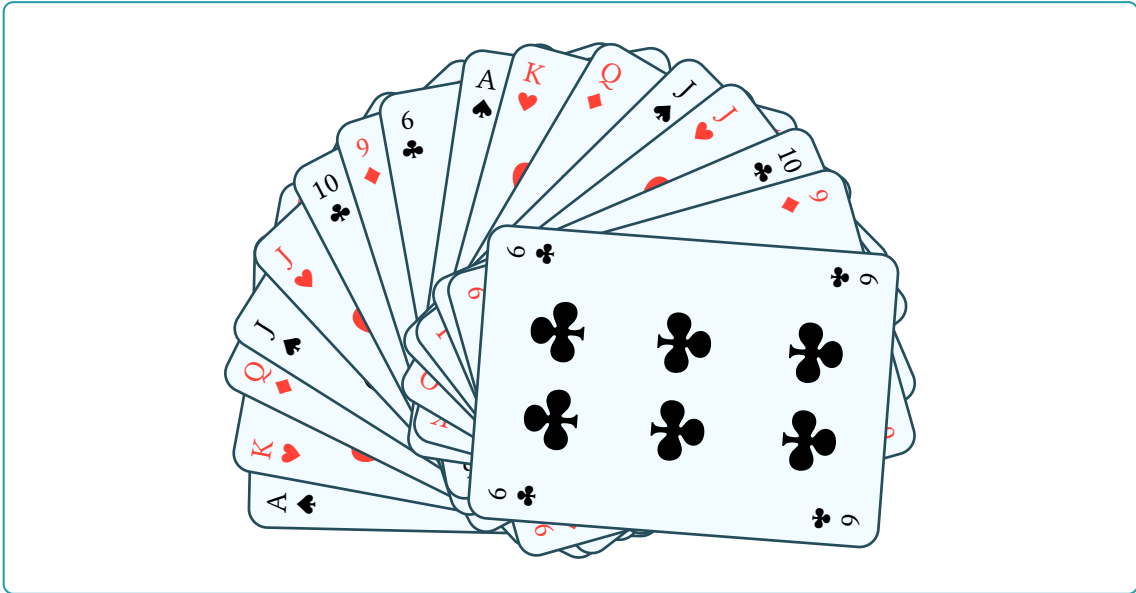
I.3.4 Heaps

DECKZ also provides a `#deckz.heap` function to display a **heap of cards**. This is similar to a hand (`#deckz.hand`), but the cards are randomly scattered within a specified area, rather than arranged in an arc. Like the hand, the heap requires a list of card identifiers as arguments, and it can be customized with various parameters.

```
#let (w, h) = (10cm, 10cm)
#box(width: w, height: h,
  fill: olive,
  stroke: olive.darken(50%) + 2pt,
)[
  #deckz.heap(format: "small", width: w, height: h, ..deckz.deck52)
]
// Note: The `deckz.deck52` array contains all 52 standard playing cards.
```
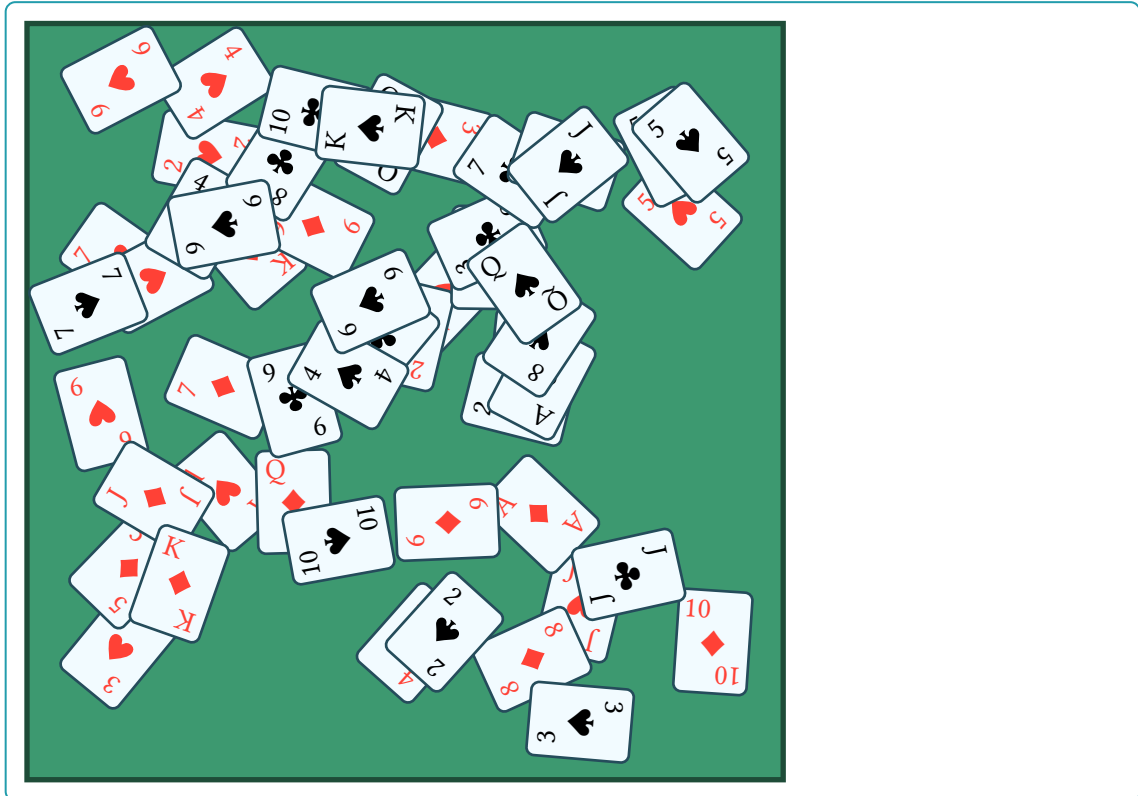
Some customization options are:
- `#deckz.heap.width` – The width of the heap, i.e. how far apart the cards will be scattered horizontally.
- `#deckz.heap.height` – The height of the heap, i.e. how far apart the cards will be scattered vertically.
- `#deckz.heap.format` – The format of the cards in the heap. It can be any of the formats described above, such as `inline`, `mini`, `small`, `medium`, `large`, or `square`. The default is `medium`.

See also Section II for the full list of parameters of `#deckz.heap`.

I.4 Customize cards

DECKZ allows for some **customization of the card appearance**, such as colors and styles.

> 🚀 *Coming Soon Feature.* This functionality is not fully supported yet: please, come back for the next releases.

I.4.1 Custom Suits

DECKZ will allow you to define **custom suits**, so you can use your own symbols or images instead of the standard hearts, diamonds, clubs, and spades.

Even though this feature is not yet implemented, you can still use custom suits by defining your own show rule for the emoji suits. In fact, DECKZ uses the emoji.suit.* symbols to render the standard suits, so you can override them with your own definitions.

For example, if you want to use a *croissant emoji* 🥐 as a custom suit instead of *diamonds* ♦, you can define it like this:

```
#show emoji.suit.diamond: text(size: 0.7em, emoji.croissant)
#deckz.hand(..deckz.cards52.diamond.values().slice(0, 7))
```
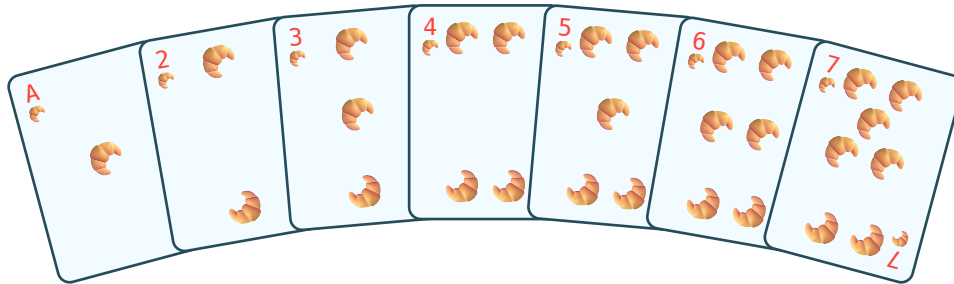
> 🔭 *Behind The Scenes.* The **resizing** of the emoji in the previous example is used to make it fit better in the card layout. When you're defining your own show rule for suits customization, you may need to adjust their size as needed.

> 🚀 *Coming Soon Feature.* Custom suits will be better supported in the future releases, allowing you to define them more easily and consistently across the deck.

I.5 Control randomness

In card games, always having the same card order, shuffle, or draw every time would be unnatural. DECKZ provides functions to manage randomness in both card rendering and manipulation, allowing for varied and realistic outcomes.

In this section, we present functions for **shuffling** or **drawing** random cards from a deck. Furthermore, randomization is also available in **rendering**: most display functions can take a noise argument to change the card's appearance slightly, making it more visually appealing and less uniform.

I.5.1 Noise in rendering

Noise is a small random variation applied to the card's rendering, which can be useful to create a more dynamic and less uniform appearance. For instance, we can slightly change a single card rendering by adding **noise** with the #deckz.render.noise argument:

```
#deckz.render("5S", format: "small") // Default: no noise
#deckz.render("5S", format: "small", noise: 0.5)
#deckz.render("5S", format: "small", noise: 1.0)
```
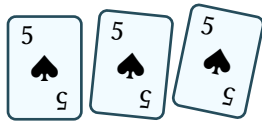

Noise can also be applied to groups of cards, such as **hands** or **decks** (see functions `#deckz.hand` and `#deckz.deck`):

```
// Hand without noise (perfectly
aligned)
#deckz.hand("AH", "KS", "QD", "JC",
format: "small", width: 4cm)

// Hand with moderate noise (slight
variations)
#deckz.hand("AH", "KS", "QD", "JC",
format: "small", width: 4cm, noise:
0.5)

// Hand with high noise (more chaotic
appearance)
#deckz.hand("AH", "KS", "QD", "JC",
format: "small", width: 4cm, noise:
1.2)
```
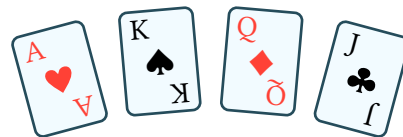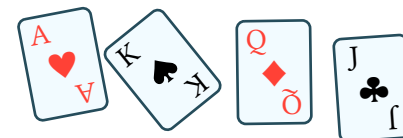

I.5.2 The true nature of random outputs

It is important to note that since *Typst* is a pure functional language, true randomness is not available. Instead, DECKZ uses the SUIJI[3] package to generate **pseudo-random numbers**, which are used to produce a deterministic – but seemingly chaotic – output.

Randomization in DECKZ is designed to be simple and intuitive: each function that requires randomness has a `rng` argument (**Random Number Generator**), which can be set to a specific value or left to the default (`auto`).

As a **general rule**:

- *If you don't provide a rng* ⇒ DECKZ will instance a **default generator** that is seeded with the current input content (usually, the cards array). This ensures that the same input will always produce the same output, making it **deterministic and reproducible**.
- *If you provide a rng* ⇒ DECKZ will use the provided generator and return the new `rng` state along with the produced output. With this approach, you can chain multiple random operations together, ensuring a pseudo-random sequence of results.

---

[3]https://typst.app/universe/package/suiji

Let's explore these two modes in detail.

I.5.3 Working with deterministic randomness

When you don't provide an RNG, DECKZ creates a deterministic output based on the input:

```
// These three calls will produce identical results
#stack(
  spacing: 5mm,
  deckz.hand("AS", "KH", "QD", format: "small", noise: 1.0, width: 5cm),
  deckz.hand("AS", "KH", "QD", format: "small", noise: 1.0, width: 5cm),
  deckz.hand("AS", "KH", "QD", format: "small", noise: 1.0, width: 5cm)
)
```
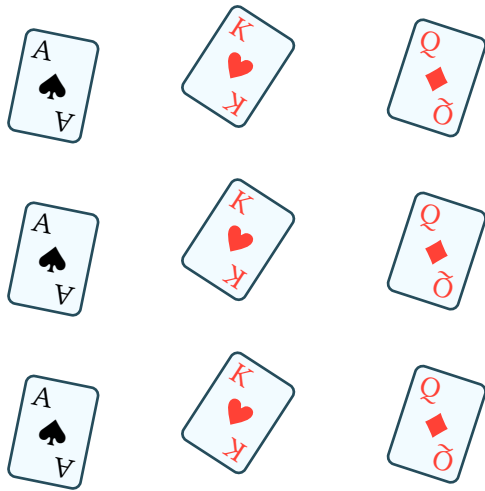

This behavior is useful for:

- *Consistent document layouts*: the same cards will always appear the same way.
- *Reproducible examples*: perfect for documentation and tutorials.
- *Version control*: documents will look identical across different compilations.

I.5.4 Creating variation with external RNGs

To get **different visual results** for the same cards, you need to provide your own RNG from outside the function. This can be done using the SUIJI[4] function `gen-rng-f(seed)`:

```
#import "@preview/suiji:0.4.0": gen-rng-f

// Create different RNGs with different seeds
#let rng1 = gen-rng-f(42)
#let rng2 = gen-rng-f(123)
#let rng3 = gen-rng-f(999)

// Same cards, different appearances
```

---

[4]https://typst.app/universe/package/suiji

```
#stack(
  spacing: 5mm,
  deckz.hand("AS", "KH", "QD", format: "small", noise: 1.0, width: 5cm, rng:
rng1).last(),
  deckz.hand("AS", "KH", "QD", format: "small", noise: 1.0, width: 5cm, rng:
rng2).last(),
  deckz.hand("AS", "KH", "QD", format: "small", noise: 1.0, width: 5cm, rng:
rng3).last(),
)
```
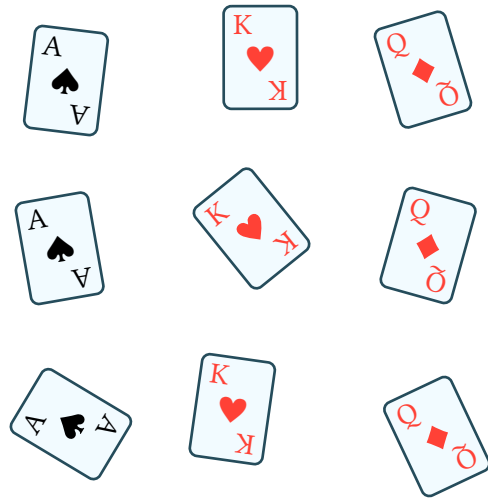

🔭 *Behind The Scenes.* You may have noticed the `.last()` at the end of each call, in the previous example. This is because when using an external RNG, DECKZ functions return a **tuple** containing the new RNG state and the rendered content. To extract the final content, you need to access the last element of the returned tuple, which is done with the `.last()` function call.

However, instead of creating separate RNGs for each call, it is often more convenient to **chain operations** using the same RNG.

```
#import "@preview/suiji:0.4.0": gen-rng-f

#let my-rng = gen-rng-f(42)

// First operation: render a single card
#let (my-rng, card1) = deckz.small("3C", noise: 0.2, rng: my-rng)

// Second operation: use the updated RNG for a different result
#let (my-rng, card2) = deckz.small("3C", noise: 0.9, rng: my-rng)

// Display both cards - they will look different!
#grid(columns: 2, gutter: 1cm, card1, card2)
```

As a general guideline, you can follow these practices when working with randomness in DECKZ:

Use deterministic mode when:	**Use external RNGs when:**
• Creating documentation;	• Simulating real game scenarios;
• Building consistent layouts;	• Creating varied visual content;
• Teaching card game rules;	• Building interactive examples;
• Making reproducible examples.	• Generating multiple variations.

I.5.5 Manipulate cards with `mix`

In addition to rendering, DECKZ provides functions for **manipulating hands and decks**, such as shuffling or drawing cards. These functions – included in the `mix` submodule of DECKZ – also support RNGs to control randomness.

Here we can see how to **shuffle** a deck of cards (`#deckz.mix.shuffle`) and **draw** a hand of five cards (`#deckz.mix.choose`) from it:

```
#import "@preview/suiji:0.4.0": gen-rng-f

// Shuffle the deck with a specific RNG
#let (my-rng, shuffled-deck) = deckz.mix.shuffle(deckz.deck52, rng: gen-rng-
f(42))

// Display the shuffled deck
The current shuffled deck: #shuffled-deck.map(deckz.inline).join()

// Draw the top 5 cards from the shuffled deck
#let (my-rng, hand) = deckz.mix.choose(shuffled-deck, n: 5, rng: my-rng)

// Display the drawn hand
We draw the following *five cards*: #deckz.hand(..hand, format: "small", width:
3cm)
```

The current shuffled deck: 2♠ 10♦ 5♦ 7♣ 10♣ 4♣ 7♦ 8♣ A♠ 6♥ A♣ 10♥ 8♠ K♦ 6♦ 2♦ K♠ J♠ Q♦ 5♥ 2♣ 9♣ 3♦ 4♥ 9♠ A♥ K♣ 5♠ 5♣ 8♦ 9♥ J♣ 6♠ 8♥ J♥ 7♠ 4♦ A♦ 3♥ 4♠ Q♠ Q♥ 9♦ 7♥ K♥ 2♥ J♦ 3♣ 10♠ Q♣ 6♣ 3♠

We draw the following **five cards**:
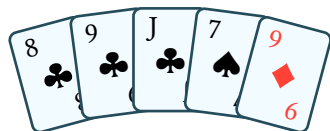
Another function included in the `mix` submodule is `#deckz.mix.split`, which allows to **split a deck** into multiple parts, according to the specified sizes. This can be useful for creating multiple hands or sub-decks from a larger deck.

```
// Randomly split the deck
#let split-deck = deckz.mix.split(deckz.deck52, size: ((3, 5), 8), rest: true)
#let ((hand1, hand2, hand3), board, remaining-cards) = split-deck

// Display the hands and the board
We split the deck into three hands and a board:
#grid(
  columns: (1fr, 1fr, 1fr),
  gutter: 1em,
  deckz.hand(..hand1, format: "small", width: 3cm),
  deckz.hand(..hand2, format: "small", width: 3cm),
  deckz.hand(..hand3, format: "small", width: 3cm),
  deckz.line(..board, format: "small", width: 10cm)
)
There are still *#remaining-cards.len() cards* left in the deck.
```

We split the deck into three hands and a board:
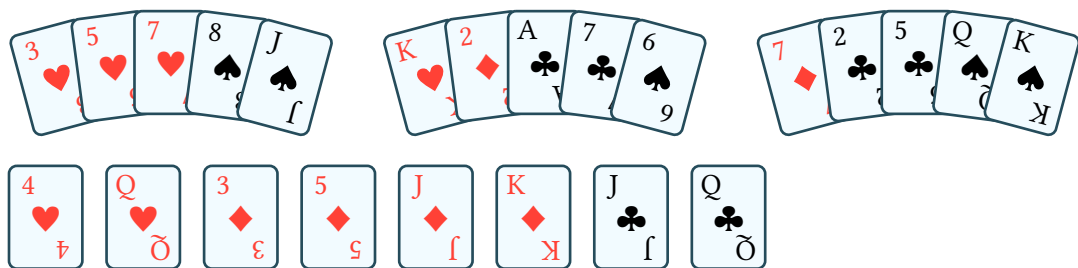
There are still **29 cards** left in the deck.

At first glance, the `#deckz.mix.split` function may seem similar to the `#deckz.mix.choose` function, but it has a fundamental difference: it keeps track of the **remaining cards** in the deck, allowing you to draw more cards later. If you use `#deckz.mix.choose` three times, you will have three different hands, but it's not guaranteed that the same card won't be drawn twice. Instead, `#deckz.mix.split` ensures that the drawn cards are unique and that you can still access the remaining cards in the deck.

> 🔭 *Behind The Scenes.* For this reason, `#deckz.mix.split` is less efficient than `#deckz.mix.choose`, as it needs to keep track of the remaining cards.

I.6 Arrange and score cards

Up to this point, all cards have been "treated equally", without regard to their suit or rank. However, in most card games, the arrangement and value of cards are crucial – winning often

depends on how cards are sorted and scored, and specific combinations can determine the outcome.

For this reason, the **arr and val sub-modules** of DECKZ include **sorting** and **scoring** functions respectively, for organizing cards and evaluating hands according to the rules of the most popular card games (especially poker).

I.6.1 Sort hands with `arr`

The `#deckz.arr.sort` function **organizes cards array by different criteria**. The resulting array of cards can be passed to a function for group visualization, such as `#deckz.hand`, to display the cards in a specific order.

```
// Define an unordered set of cards
#let cards = ("AS", "KH", "QD", "JC", "10S", "2H")

// Sort by standard order (suit first, then rank)
#deckz.line(..deckz.arr.sort(cards, by: "order"), format: "small")

// Sort by score (high cards first: A, K, Q, J, 10...)
#deckz.line(..deckz.arr.sort(cards, by: "score"), format: "small")
```
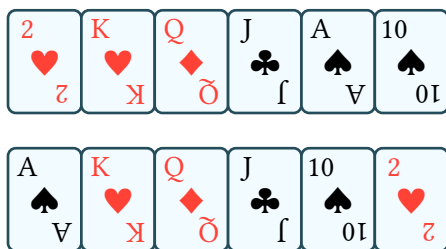

Sometimes it can be useful to **count or group cards in the hand** according to their suit or rank, for example to display them in a more organized way. The `#deckz.arr.group-ranks` and `#deckz.arr.group-suits` functions can be used for this purpose.

```
#let hand = ("AS", "AH", "KS", "KH", "QD", "5H")

// Count occurrences of each rank
Count the number of each rank in a hand:
#deckz.arr.count-ranks(hand)

// Group cards by rank or suit
Group cards by *rank*:
#for (rank, cards) in deckz.arr.group-ranks(hand).pairs() [
  - #rank: #cards
]
Group cards by *suit*:
#for (suit, cards) in deckz.arr.group-suits(hand).pairs() [
```

```
    - #suit: #cards
]
```

Count the number of each rank in a hand: (ace: 2, king: 2, queen: 1, five: 1)

Group cards by **rank**:
- ace: ("AS", "AH")
- king: ("KS", "KH")
- queen: ("QD",)
- five: ("5H",)

Group cards by **suit**:
- spade: ("AS", "KS")
- heart: ("AH", "KH", "5H")
- diamond: ("QD",)

I.6.2 Score hands with `val`

DECKZ includes functions to **evaluate hands of cards** according to the rules of the most common card games. More specifically, it is possible to assess **poker hands** with three types of functions:

- **Detection**: check if a hand *contains* a specific combinations.
- **Validation**: check if a hand *exactly matches* a specific combination.
- **Hand Extraction**: extract *all possible combinations* of a specific type from a hand.

🔭 *Behind The Scenes.* The combinations recognized by DECKZ are the standard poker combinations:

Combination	Description	Example
high-card	The singular highest card of the hand	3♠
pair	A pair of cards with the same rank	3♠ 3♥
two-pairs	Two pairs of cards with the same rank	3♠ 3♥ K♠ K♥
three-of-a-kind	Three cards of the same rank	9♥ 9♠ 9♣
straight	Five cards in a sequence, regardless of suit	7♠ 8♠ 9♥ 10♦ J♣
flush	Five cards of the same suit	3♠ K♠ 9♠ 4♠ 10♠
full-house	Three cards of one rank and two cards of another	9♥ 9♠ 9♣ 3♠ 3♥
four-of-a-kind	Four cards of the same rank	

		4♠ 4♥ 4♣ 4♦
straight-flush	Five cards in a sequence, all of the same suit	7♠ 8♠ 9♠ 10♠ J♠
five-of-a-kind	Five cards of the same rank (only possible with multiple copies of the same card)	A♠ A♠ A♥ A♣ A♦

Detection and validation

DECKZ offers many functions to check if a hand **contains** (has-X) or **exactly matches** (is-X) a specific combination X (where X is one of the combinations defined in the table above).

Detection and validation functions return a bool value, and can be used as follows:

```
#let cards = ("AS", "AH", "KD", "QC", "JH")
#deckz.val.has-pair(cards) // true (contains a pair)
#deckz.val.is-pair(cards) // false (5 cards, not exactly 2)
#deckz.val.is-pair(("AS", "AH")) // true (exactly 2 matching cards)
```

```
true false true
```

Extraction of a given combination

DECKZ offers many functions to **extract** all possible combinations of a specific type from a hand. The return type is an array of all combinations found, which can be used to display the cards in a specific order or to evaluate the hand.

```
#let available-cards = ("AS", "AH", "KS", "KH", "QD")

// Get all possible pairs from these cards
All possible pairs:
#for pair in deckz.val.extract-pair(available-cards) [ - #pair ]

// Find the best hand automatically
Best hand combination:
#deckz.val.extract-highest(available-cards).first()
```

All possible pairs:
- ("AS", "AH")
- ("KS", "KH")

Best hand combination: ("AS", "AH", "KS", "KH")

The general #deckz.val.extract function can be used to extract any combination specified with the #deckz.val.extract.scoring-combination parameter.

```
#let some-cards = ("AS", "3H", "KS", "3C", "2C", "3S", "6H", "7H", "3D")

// These are equivalent:
Extracted combinations:
#deckz.val.extract("three-of-a-kind", some-cards)

Extracted combinations:
#deckz.val.extract-three-of-a-kind(some-cards)
```

Extracted combinations: (
```
  ("3H", "3C", "3S"),
  ("3H", "3C", "3D"),
  ("3H", "3S", "3D"),
  ("3C", "3S", "3D"),
)
```
Extracted combinations: (
```
  ("3H", "3C", "3S"),
  ("3H", "3C", "3D"),
  ("3H", "3S", "3D"),
  ("3C", "3S", "3D"),
)
```

🔭 *Behind The Scenes.* All extraction functions are **stable functions**, meaning they evaluate the hand in a consistent order: if two cards have a certain order, their relative order will be preserved in the output. This is particularly useful for games where the order of cards matters.

Part II
Documentation

II.1 Card visualization

This section provides a comprehensive overview of the DECKZ package's **card visualization** capabilities. It presents the available formats and how to use them effectively.

```
#deckz.back          #deckz.medium        #deckz.small
#deckz.inline        #deckz.mini          #deckz.square
#deckz.large         #deckz.render
```

#deckz.back(⟨format⟩: "medium") → `content`

Renders the back of a card in the specified format. Currently, it only supports one style, which is a simple rectangle with a rhombus pattern.

```
#deckz.back(format: "medium")
#deckz.back(format: "small")
#deckz.back(format: "mini")
#deckz.back(format: "inline")
```
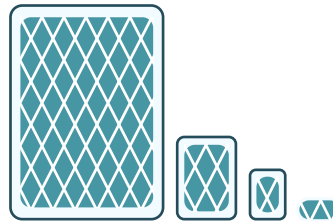

> **Argument**
>
> ⟨format⟩: "medium"   `str`
>
> The format of the card back, defaults to `"medium"`. Available formats: `"inline"`|`"mini"`|`"small"`|`"medium"`|`"large"`|`"square"`.

#deckz.inline(⟨card⟩, ..⟨args⟩) → `content`

Renders a card with the "**inline**" format. The card is displayed in a compact style: text size is coherent with the surrounding text, and the card is rendered as a simple text representation of its rank and suit.

```
#lorem(10)
#deckz.inline("AS"), #deckz.inline("3S")
#lorem(10)
#deckz.inline("KH").
```

```
Lorem ipsum dolor sit amet,
consectetur adipiscing elit, sed do.
A♠ , 3♠ Lorem ipsum dolor sit amet,
consectetur adipiscing elit, sed do.
K♥ .
```

This is a wrapper around the `#deckz.render` function with the format set to "inline". Every additional argument passed to this function will be forwarded to the `#deckz.render` function.

> **Argument**
>
> ⟨card⟩   `str`

> The code of the card you want to represent.

```
┌─ Argument ─────────────────────────────────────────────────────────┐
  ..(args)                                                        any
  
  Additional arguments to pass to the rendering function #deckz.render.
└─────────────────────────────────────────────────────────────────────┘
```

#deckz.large((card), ..(args)) → `content`

Renders a card with the "**large**" format, emphasizing the card's details: all four corners are used to display the rank and suit, with a large central representation. Like other formats, the large format is responsive to text size; corner summaries are scaled accordingly to the current text size.

```
#deckz.large("JD")
#deckz.large("9C")
```
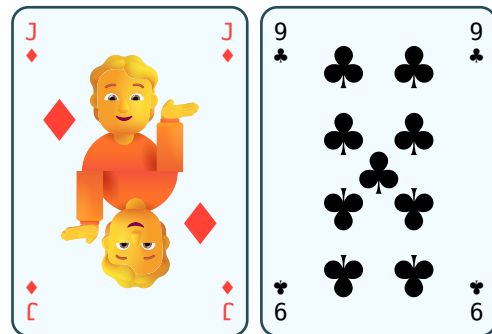

This is a wrapper around the #deckz.render function with the format set to "large". Every additional argument passed to this function will be forwarded to the #deckz.render function.

```
┌─ Argument ─────────────────────────────────────────────────────────┐
  (card)                                                          str
  
  The code of the card you want to represent.
└─────────────────────────────────────────────────────────────────────┘
```

```
┌─ Argument ─────────────────────────────────────────────────────────┐
  ..(args)                                                        any
  
  Additional arguments to pass to the rendering function #deckz.render.
└─────────────────────────────────────────────────────────────────────┘
```

#deckz.medium((card), ..(args)) → `content`

Renders a card with the "**medium**" format: a full, structured card layout with two corner summaries and realistic suit placement. The medium format is usually the default format for card rendering in DECKZ.

```
#deckz.medium("QD")
#deckz.medium("AH")
#deckz.medium("7C")
```
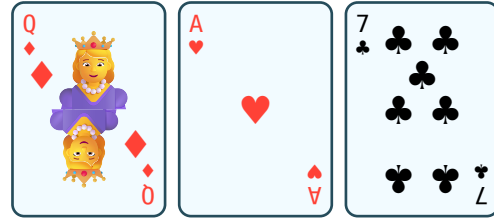
This is a wrapper around the #deckz.render function with the format set to "medium". Every additional argument passed to this function will be forwarded to the #deckz.render function.

─ Argument ─
〈card〉   str

The code of the card you want to represent.

─ Argument ─
..〈args〉   any

Additional arguments to pass to the rendering function #deckz.render.

#deckz.mini(〈card〉, ..〈args〉) → content

Renders a card with the "**mini**" format. The card is displayed in a very compact style, suitable for dense layouts. The frame size is responsive to text, and it contains a small representation of the card's rank and suit.

```
#deckz.mini("JC")
#deckz.mini("AH")
#deckz.mini("5S")
#deckz.mini("9D")
#deckz.mini("4H")
#deckz.mini("3C")
#deckz.mini("2D")
#deckz.mini("KS")
```

This is a wrapper around the #deckz.render function with the format set to "mini". Every additional argument passed to this function will be forwarded to the #deckz.render function.

─ Argument ─
〈card〉   str

The code of the card you want to represent.

─ Argument ─
..〈args〉   any

Additional arguments to pass to the rendering function #deckz.render.

#deckz.render(〈card〉, 〈format〉: "medium", 〈noise〉: none, 〈rng〉: auto) → content

Render function to view cards in different formats. This function allows you to specify the format of the card to be rendered. Available formats include: inline, mini, small, medium, large, and square.

```
#deckz.render("10S")
#deckz.render(format: "large", "8C")
#deckz.render(format: "mini", "KH")
```
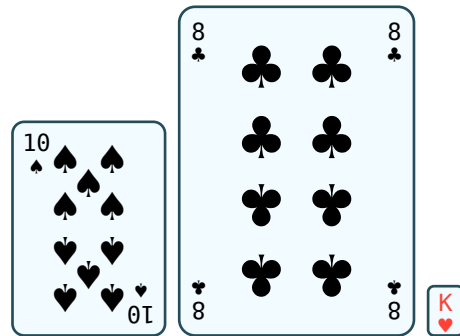
┌─ Argument ───┐
│ `(card)`   `str` │
│   │
│   The code of the card you want to represent.   │
└───┘

┌─ Argument ───┐
│ `(format): "medium"`   `str` │
│   │
│   The   selected   format   of   the   card.   Available   formats   are: │
│   `"inline"`\|`"mini"`\|`"small"`\|`"medium"`\|`"large"`\|`"square"`.   │
└───┘

┌─ Argument ───┐
│ `(noise): none`   `float` \| `none` │
│   │
│   The amount of "randomness" in the placement and rotation of the card. Default value │
│   is `none` or `0.0`, which corresponds to no variations. A value of `1.0` corresponds to a │
│   "standard" amount of noise, according to DECKZ style. Higher values might produce │
│   crazy results, handle with care.   │
└───┘

┌─ Argument ───┐
│ `(rng): auto`   `rng` \| `auto` │
│   │
│   The random number generator to use for the noise. If not provided or set to default │
│   value `auto`, a new random number generator will be created. Otherwise, you can pass │
│   an existing random number generator to use.   │
└───┘

#**deckz**.**small**(`(card)`, ..`(args)`) → `content`

Renders a card with the "**small**" format. The card is displayed in a concise style, balancing readability and space: the card's rank is shown symmetrically in two corners, with the suit displayed in the center.

```
#deckz.small("3S")
#deckz.small("6H")
#deckz.small("QS")
#deckz.small("5D")
#deckz.small("AC")
#deckz.small("4S")
```

This is a wrapper around the #deckz.render function with the format set to "small". Every additional argument passed to this function will be forwarded to the #deckz.render function.

> — Argument —
> (card)   str
>
> The code of the card you want to represent.

> — Argument —
> ..(args)   any
>
> Additional arguments to pass to the rendering function #deckz.render.

#deckz.square((card), ..(args)) → content

Renders a card with the "**square**" format, i.e. with a frame layout with 1:1 ratio. This may be useful for grid layouts or for situations where the cards are often rotated in many directions, because the corner summaries are placed diagonally.

```
#deckz.square("5C")
#deckz.square("JH")
```
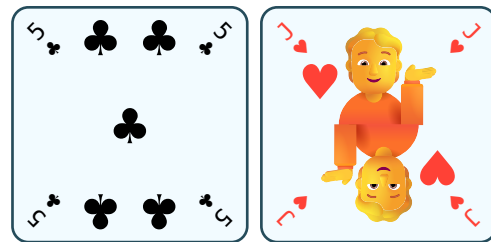
This is a wrapper around the #deckz.render function with the format set to "square". Every additional argument passed to this function will be forwarded to the #deckz.render function.

> — Argument —
> (card)   str
>
> The code of the card you want to represent.

> — Argument —
> ..(args)   any
>
> Additional arguments to pass to the rendering function #deckz.render.

II.2 Group visualization

This section covers the **group visualization** features of the DECKZ package, i.e. all functions that allow you to visualize groups of cards, such as hands, decks, and heaps.

```
#deckz.deck                    #deckz.heap
#deckz.hand                    #deckz.line
```

#deckz.deck(
  (top-card),
  (format): "medium",
  (angle): 60deg,
  (height): 1cm,
  (noise): none,
  (rng): auto
) → `content`

Renders a **stack** of cards, as if they were placed one ontop of each other. Calculates the number of cards based on the given height (`#deckz.deck.height`), and spaces them evenly along the specified angle (`#deckz.deck.angle`). Each card is rendered with a positional shift to create a fanned deck appearance.

```
#deckz.deck(
  angle: 20deg,
  height: 1.5cm,
  "7D"
)
```
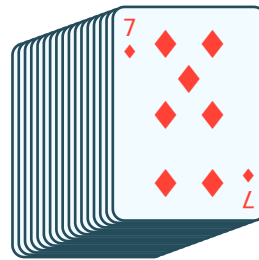
Argument

`(top-card)` `str`

  The **top card** in the deck, with standard code representation.

Argument

`(format): "medium"` `str`

  The **format** to use for rendering each card. Default is "medium".

Argument

`(angle): 60deg` `angle`

  The **angle** at which the deck is fanned out. Default is `60deg`.

Argument

`(height): 1cm` `height`

> The total **height** of the deck stack. This determines how many cards are rendered in the stack, as one card is displayed for every `2.5pt` of height.

┌─ Argument ───┐
│ `(noise): none`   `float` `none` │
│   │
│ The amount of "**randomness**" in the placement and rotation of the card. Default value is "none" or "0", which corresponds to no variations. A value of 1 corresponds to a "standard" amount of noise, according to Deckz style. Higher values might produce crazy results, handle with care. │
└───┘

┌─ Argument ───┐
│ `(rng): auto`   `rng` `auto` │
│   │
│ The **random number generator** to use for the noise. If not provided or set to default value `auto`, a new random number generator will be created. Otherwise, you can pass an existing random number generator to use. │
└───┘

```
#deckz.hand(
  ..(cards),
  (format): "medium",
  (angle): 30deg,
  (width): 10cm,
  (noise): none,
  (rng): auto
) → content
```

Displays a **sequence of cards** in a horizontal hand layout and applying a slight rotation to each card, creating an arched effect.

This function is useful for displaying a hand of cards in a visually appealing way. It accepts any number of cards, each represented by a string identifier (e.g., `"AH"` for Ace of Hearts).

```
#deckz.hand(
  width: 100pt,
  "AH", "AD", "AS", "AC"
  // Poker of Aces
)
```
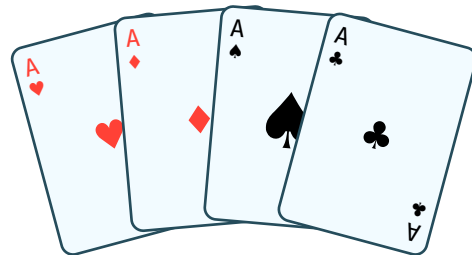

┌─ Argument ───┐
│ `..(cards)`   `array` │
│   │
│ The list of **cards** to display, with standard code representation. │
└───┘

```
┌─ Argument ─────────────────────────────────────────────────────────────────┐
│ (format): "medium"                                                      str │
│                                                                             │
│   The format of the cards to render. Default is "medium". Available formats: inline, │
│   mini, small, medium, large, square.                                       │
└─────────────────────────────────────────────────────────────────────────────┘
```

```
┌─ Argument ─────────────────────────────────────────────────────────────────┐
│ (angle): 30deg                                                        angle │
│                                                                             │
│   The angle between the first and last card, i.e. the angle covered by the arc. │
└─────────────────────────────────────────────────────────────────────────────┘
```

```
┌─ Argument ─────────────────────────────────────────────────────────────────┐
│ (width): 10cm                                                        length │
│                                                                             │
│   The width of the hand, i.e. the distance between the first and last card. │
└─────────────────────────────────────────────────────────────────────────────┘
```

```
┌─ Argument ─────────────────────────────────────────────────────────────────┐
│ (noise): none                                                 float │ none  │
│                                                                             │
│   The amount of "randomness" in the placement and rotation of the card. Default value │
│   is "none" or "0", which corresponds to no variations. A value of "1" corresponds to a │
│   "standard" amount of noise, according to DECKZ style. Higher values might produce │
│   crazy results, handle with care.                                          │
└─────────────────────────────────────────────────────────────────────────────┘
```

```
┌─ Argument ─────────────────────────────────────────────────────────────────┐
│ (rng): auto                                                     rng │ auto  │
│                                                                             │
│   The random number generator to use for the noise. If not provided or set to default │
│   value auto, a new random number generator will be created. Otherwise, you can pass │
│   an existing random number generator to use.                               │
└─────────────────────────────────────────────────────────────────────────────┘
```

```
#deckz.heap(
  ..(cards),
  (format): "medium",
  (width): 10cm,
  (height): 10cm,
  (exceed): false,
  (rng): auto
) → content
```

Renders a **heap** of cards, randomly placed in the given area. The cards are placed in a random position within the specified width (`#deckz.heap.width`) and height (`#deckz.heap.height`), with a random rotation applied to each card. The `#deckz.heap.exceed` parameter controls whether cards can exceed the specified frame dimensions or not.

```
#deckz.heap(
  format: "small",
  width: 5cm,
  height: 4cm,
  "7D", "8D", "9D", "10D", "JD"
)
```
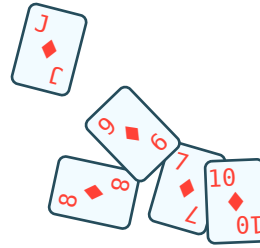
— Argument —

`..(cards)`   `array`

The **cards to display**, with standard code representation. The last cards are repre-
sented on top of the previous one, as the rendering follows the given order.

— Argument —

`(format): "medium"`   `str`

The **format** to use for rendering each card. Default is "medium".

— Argument —

`(width): 10cm`   `length`

The **horizontal dimension** of the area in which cards are placed.

— Argument —

`(height): 10cm`   `length`

The **vertical dimension** of the area in which cards are placed.

— Argument —

`(exceed): false`   `bool`

If `true`, allows cards to **exceed the frame** with the given dimensions. When the
parameter is `false`, instead, cards placement considers a margin of half the card length
on all four sides. This way, it is guaranteed that cards are placed within the specified
frame size. Default is `false`.

```
// Example with `exceed: true`
#box(width: 3cm, height: 3cm, stroke:
green)[
  #place(center + horizon, deckz.heap(
    format: "small",
    width: 3cm,
    height: 3cm,
    exceed: true,
    ..deckz.deck52.slice(0, 13)
  ))
]

// Example with `exceed: false`
#box(width: 3cm, height: 3cm, stroke:
green)[
  #place(center + horizon, deckz.heap(
    format: "small",
    width: 3cm,
    height: 3cm,
    exceed: false,
    ..deckz.deck52.slice(0, 13)
  ))
]
```
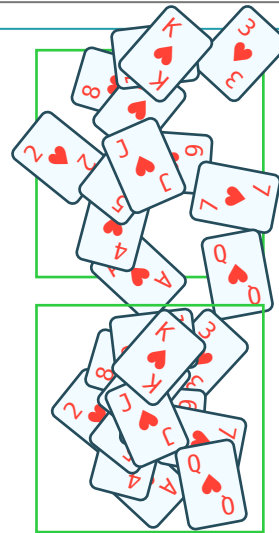

---- Argument ----

(rng): auto   rng | auto

The **random number generator** to use for cards displacement. If not provided or
set to default value `auto`, a new random number generator will be created. Otherwise,
you can pass an existing random number generator to use.

```
#deckz.line(
  ..(cards),
  (format): "medium",
  (spacing): auto,
  (width): auto,
  (noise): none,
  (rng): auto
) → content
```

Displays a **sequence of cards** in a horizontal line layout. The function accepts any number
of cards, each represented by a string identifier (e.g., `"AH"` for Ace of Hearts). It accepts
parameters to control the format of the cards and the area they occupy. The value of `spacing`
determines the space between adjacent cards, while `width` specifies the total width of the line.
Important: being one dependent on the other, only one parameter between `spacing` and
`width` can be set to a specific value at a time. If none of them is set, cards are placed adjacent
with no extra space.

```
#let hand = ("AH", "AD", "AS", "AC")

#deckz.line(..hand, format: "small")

#deckz.line(..hand, format: "small", spacing:
10pt)

#deckz.line(..hand, format: "small", width:
6cm)

#deckz.line(..hand, format: "small", width:
100%)

#box(width: 6cm, stroke: purple)[
  #deckz.line(..hand, format: "small", width:
100%)
]
```
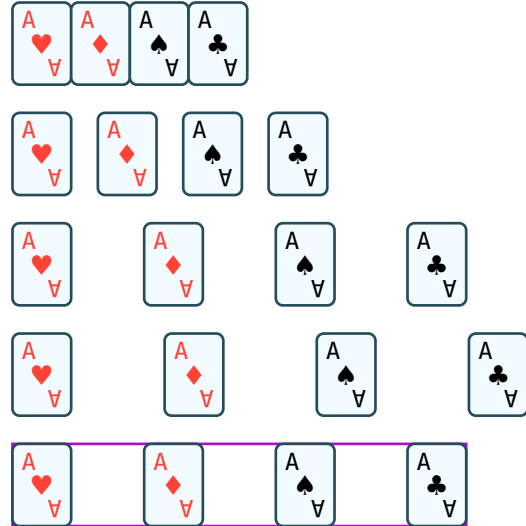
— Argument —

`..(cards)`   `array`

The list of **cards** to display, with standard code representation.

— Argument —

`(format): "medium"`   `str`

The **format** of the cards to render. Default is "medium". Available formats: `inline`,
`mini`, `small`, `medium`, `large`, `square`.

— Argument —

`(spacing): auto`   `length` | `auto` | `none`

The **spacing** between adjacent cards.
- If set, cards are distributed with the specified spacing.
- Default is `auto`. If left to default value, cards are placed adjacent with no extra space.
- In any other case – i.e. when `spacing` is set to a specific positive non-zero value,
  the function requires `width` to be left as `auto` or set to `none` otherwise, the function
  will panic.

— Argument —

`(width): auto`   `length` | `auto` | `none`

The **total width** of the line, measured from the center of the first card to the center
of the last card.
- If set, cards are distributed evenly along this width.
- Requires `spacing` to be left as `auto` or set to `none` otherwise, the function will panic.
- Default is `auto`. If both `width` and `spacing` are `auto`, cards are placed adjacent with
  no extra space.

---- Argument ----

`(noise): none`   `float` | `none`

The amount of "**randomness**" in the placement and rotation of the card. Default value is `none` or `0`, which corresponds to no variations. A value of `1` corresponds to a "standard" amount of noise, according to DECKZ style. Higher values might produce crazy results, handle with care.

---- Argument ----

`(rng): auto`   `rng` | `auto`

The **random number generator** utilized for the noise. If not provided or set to default value `auto`, a new random number generator will be created. Otherwise, you can pass an existing random number generator to use.

II.3 Data

This section provides an overview of the **data structures** used in the DECKZ package, including suits, ranks, and cards. It explains how these data structures are organized and how to access them.

#suits

A mapping of all **suit symbols** utilized in DECKZ.

Primarily intended for internal use within higher-level functions, but can also be accessed directly, for example, to iterate over the four suits.

```
#stack(                                    ♥  ♦  ♣  ♠
  dir: ltr,
  spacing: 1em,
  ..deckz.suits.values().map(suit-data => {
    text(suit-data.color)[#suit-data.symbol]
  })
)
```

#ranks

A mapping of all **rank symbols** utilized in DECKZ.

This dictionary is primarily intended for internal use within higher-level functions, but can also be accessed directly, for example, to iterate over the ranks.

```
#table(
  columns: 5 * (1fr, ),
  ..deckz.ranks.keys()
)
```

ace	two	three	four	five
six	seven	eight	nine	ten
jack	queen	king		

#cards52

This is a dictionary of **all the cards in a deck**.

It is structured as cards.<suit-k>.<rank-k>, where:
- <suit-k> is one of the keys from the suits dictionary, and
- <rank-k> is one of the keys from the ranks dictionary.

The value associated with each (rank, suit) pair is the **card code**, which is a string in the format <rank-s><suit-s>, where <rank-s> is the rank symbol and <suit-s> is the first letter of the suit key in uppercase.

```
#deckz.cards52.heart.ace // Returns "AH"      AH KS 10D 3C
#deckz.cards52.spade.king // Returns "KS"
#deckz.cards52.diamond.ten // Returns "10D"
#deckz.cards52.club.three // Returns "3C"
```

This dictionary can be used to access any card in a standard deck of 52 playing cards by its suit and rank, and use it in various functions that require a card code.

Here is an example using the #deckz.hand function:

```
#deckz.hand(
  format: "small",
  width: 128pt,
  angle: 90deg,
  ..deckz.cards52.heart.values(),
)
```

#deck52   `array`

A list of all the cards in a standard deck of 52 playing cards. It is a *flat* list of **card codes**, where each code is a string in the format <rank-s><suit-s>, where <rank-s> is the rank symbol and <suit-s> is the first letter of the suit key in uppercase. It is created programmatically from the suits and ranks dictionaries.

```
#table(
  columns: 13,
  align: center,
  stroke: none,
  ..deckz.deck52,
)
```

AH	2H	3H	4H	5H	6H	7H	8H	9H	10H	JH	QH	KH
AD	2D	3D	4D	5D	6D	7D	8D	9D	10D	JD	QD	KD
AC	2C	3C	4C	5C	6C	7C	8C	9C	10C	JC	QC	KC
AS	2S	3S	4S	5S	6S	7S	8S	9S	10S	JS	QS	KS

This array can be used in various functions that require a list of card codes, such as #deckz.hand, #deckz.deck, or #deckz.heap.

Here is an example using the #deckz.heap function:

```
#deckz.heap(
  format: "small",
  width: 128pt,
  height: 10cm,
  ..deckz.deck52.slice(26, 52),
)
```
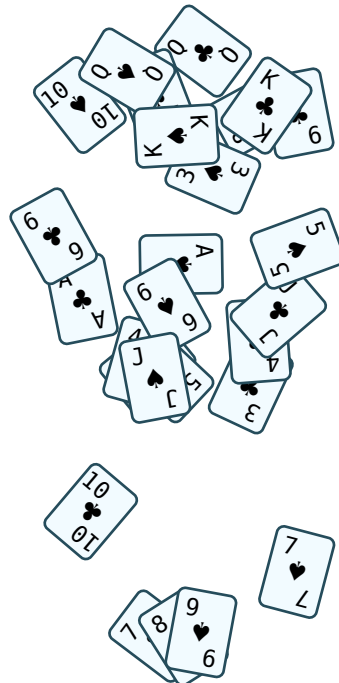
II.4 Randomization

The ᴅᴇᴄᴋᴢ package includes essential **randomization features** for card games, such as shuffling and drawing random cards from a deck. However, since **Typst** is a pure functional language, true randomness is not available; instead, ᴅᴇᴄᴋᴢ uses the ꜱᴜɪᴊɪ[5] package to generate pseudo-random numbers.

This section explains how to use these randomization tools within ᴅᴇᴄᴋᴢ, describes the underlying concepts, and provides practical guidance for integrating randomness into your projects. Randomization functions are available under the `deckz.mix` namespace, and they include:

```
#deckz.mix.choose          #deckz.mix.shuffle          #deckz.mix.split
```

```
#deckz.mix.choose(
  (cards),
  (n): 1,
  (replacement): false,
  (permutation): false,
  (rng): auto
) → array
```

Choose `n` cards from the given array of cards, optionally allowing replacement and permutation. This is useful for drawing random cards from a deck or selecting cards from a hand.

The function behaves as follows in case of different values of `n`:
- If `n` is 1 (the default), it will return a single card chosen randomly from the given cards. The card will be contained in an array of length 1.
- If `n` is greater than 1, it will return an array of `n` cards chosen randomly from the given cards.
  - If `replacement` is `true`, the same card can be chosen multiple times. This means that the function can return duplicates in the resulting array.
- If `n` is greater than the number of cards and `replacement` is `false`, the function will return all cards (optionally shuffled, if `permutation` is `true`). This corresponds to drawing all cards from the deck.
- If `n` is 0 or less, the function will return an empty array.

This function uses the `suiji.choose-f` function from the ꜱᴜɪᴊɪ[6] library to draw elements from the array. As such, it accepts a **random number generator** (`rng`) as an optional parameter (more information in the `#deckz.mix.choose.rng` parameter documentation).

The return value of this function depends on the `rng` parameter:

- If `rng` is set to `auto`, the function will return only the chosen cards, i.e. an array of cards.
- If `rng` is set to a specific `rng`, the function will return a tuple containing the updated `rng` and the chosen cards. *(This is useful if you want to use the same rng across multiple calls involving pseudo-randomness, or if you want to use a specific random number generator that you have already created.)*

---

[5]https://typst.app/universe/package/suiji
[6]https://typst.app/universe/package/suiji

```
┌─ Argument ─────────────────────────────────────────────────┐
│ (cards)                                              array  │
│                                                            │
│   The cards to choose from.                                │
└────────────────────────────────────────────────────────────┘
```

```
┌─ Argument ─────────────────────────────────────────────────┐
│ (n): 1                                                 int  │
│                                                            │
│   The number of cards to choose. Default is 1, meaning     │
│   that only one card will be chosen from the given cards.  │
└────────────────────────────────────────────────────────────┘
```

The number of cards to choose. Default is 1, meaning that only one card will be chosen from the given cards.

If n is greater than the number of cards and `replacement` is `false`, the function will return an error. If `replacement` is `true`, the function will return n cards, possibly including duplicates. If n is 0 or less, the function will return an empty array.

```
┌─ Argument ─────────────────────────────────────────────────┐
│ (replacement): false                                  bool │
└────────────────────────────────────────────────────────────┘
```

Whether to allow replacement of cards. If `true`, the same card can be chosen multiple times. If `false`, each card can only be chosen once.

```
┌─ Argument ─────────────────────────────────────────────────┐
│ (permutation): false                                  bool │
└────────────────────────────────────────────────────────────┘
```

Whether the sample is permuted when choosing. If `true`, the order of the chosen cards is random. If `false`, the order of the chosen cards is the same as in the original array. According to the `suiji` library, false provides a faster implementation, but the order of the chosen cards will not be random.

```
┌─ Argument ─────────────────────────────────────────────────┐
│ (rng): auto                                   (suiji.rng)  │
└────────────────────────────────────────────────────────────┘
```

The **random number generator** (rng) to use, from the SUIJI[7] package.

- **Default value** is `auto`, which corresponds to creating a new `rng` internally and automatically, using a seed derived from the cards. *(This means that the choice will be deterministic based on the input cards, and the same cards will always produce the same selection.)* Using the `auto` value is recommended for most use cases, especially when this is the only time you need a pseudo-random number generator in your code.
- The other option is to provide an **externally-defined** `rng`, which will be used and updated according to the Suiji library's conventions. This is useful when you want to use the same `rng` across multiple calls, or when you want to use a specific random number generator that you have already created.

The value of this parameter also influences the return value of the function:
- If `rng` is set to `auto`, the function will return **only the chosen cards**, i.e. an array of cards.

---

[7]https://typst.app/universe/package/suiji

> - If `rng` is set to a specific `rng`, the function will return **a tuple** containing the updated `rng` and the chosen cards.

`#deckz.mix.shuffle(⟨cards⟩, ⟨rng⟩: auto)` → `array`

Shuffle the given cards and return the **shuffled array**, i.e. a new array with the same elements in a different (random) order.

To handle randomness, this function accepts a **random number generator** (`rng`) as an optional parameter. This `rng` follows the conventions of the SUIJI[8] library, which provides a random number generator. See the `#deckz.mix.shuffle.rng` parameter documentation for more information.

This function also relies on the `suiji.shuffle-f` function from the SUIJI[9] library to shuffle the cards.

The return value of this function depends on the `rng` parameter:

- If `rng` is set to `auto`, the function will return only the shuffled cards, i.e. an array of cards in a random order.
- If `rng` is set to a specific `rng`, the function will return a tuple containing the updated `rng` and the shuffled cards. *(This is useful if you want to use the same rng across multiple calls to the `shuffle` function, or if you want to use a specific random number generator that you have already created.)*

┌─ Argument ───┐
│ `⟨cards⟩`   `array` │
│   │
│   The cards to shuffle.   │
└───┘

┌─ Argument ───┐
│ `⟨rng⟩: auto`   `(suiji.rng)` │
│   │
│   The **random number generator** (`rng`) to use, from the SUIJI[10] package. │
│   │
│   - **Default value** is `auto`, which corresponds to creating a new `rng` internally and automatically, using a seed derived from the cards. *(This means that the shuffling will be deterministic based on the input cards, and the same cards will always produce the same shuffled order.)* Using the `auto` value is recommended for most use cases, especially when this is the only time you need a pseudo-random number generator in your code. │
│   - The other option is to provide an **externally-defined** `rng`, which will be used and updated according to the Suiji library's conventions. This is useful when you want to use the same `rng` across multiple calls to the `shuffle` function, or when you want to use a specific random number generator that you have already created. │
└───┘

---

[8]https://typst.app/universe/package/suiji
[9]https://typst.app/universe/package/suiji
[10]https://typst.app/universe/package/suiji

> The value of this parameter also influences the return value of the function:
> - If `rng` is set to `auto`, the function will return **only the shuffled cards**.
> - If `rng` is set to a specific `rng`, the function will return **a tuple** containing the updated `rng` and the shuffled cards.

#deckz.mix.split((cards), (size): 1, (rest): true, (rng): auto) → `array`

Split the given deck of cards into groups of specified sizes. This is useful for dealing cards to players or creating smaller decks from a larger one.

The split is governed by the #deckz.mix.split.size parameter, which can be a *single integer* (if only one group of cards should be split from the original deck array) or an *array of integers* (if multiple groups should be dealt from the same original deck array). With this function, cards are guaranteed to be dealt **in-place**, meaning that the order of the cards in the original deck is preserved in the output groups. If you want to shuffle the cards before splitting them, you can use the #deckz.mix.shuffle function before calling this function.

The function produces an array containing **the groups of cards**, with some important warnings:
- if the #deckz.mix.split.rest is `true`, the returned array will contain all cards in the input deck, with the remaining cards (i.e. those which were not split into groups of the specified sizes) at the end of the array, in one single group.
- if the #deckz.mix.split.rest is `false`, the returned array will contain only the split cards, and those which are not contained in the split groups will be discarded.

Also, the function accepts a **random number generator** (`rng`) as an optional parameter. This `rng` follows the conventions of the SUIJI[11] library, which provides a random number generator. See the #deckz.mix.split.rng parameter documentation for more information.

The return value of this function depends on the `rng` parameter:
- If `rng` is set to `auto`, the function will return only the split groups, i.e. an array of arrays of cards.
- If `rng` is set to a specific `rng`, the function will return a tuple containing the updated `rng` and the split groups. *(This is useful if you want to use the same rng across multiple calls involving pseudo-randomness, or if you want to use a specific random number generator that you have already created.)*

Argument

`(cards)`   `array`

  The deck of cards to split.

Argument

`(size): 1`   `int | array`

  The size of the groups that the deck will be split into. This parameter can accept:
  - An **integer**, which will be used as the size of only one group.

---

[11]https://typst.app/universe/package/suiji

```
1   #deckz.split(cards, size: 5) // Splits the deck into a group of 5
    cards, and the rest of the cards.
```

- An **array of sizes** (where "sizes" are integers or array of integers), which will be used as the sizes of the groups.

```
1   #deckz.split(cards, size: (5, 3, 2)) // Splits the deck into three
    groups of 5, 3, and 2 cards respectively, and the rest of the
    cards in a fourth group.

2   #deckz.split(cards, size: (5, (3, 2))) // Splits the deck into a
    first group of 5 cards, a second group with of 6 cards structured
    as a _bidimensional matrix_ of dimensions 3x2, and a last group
    with the remaining cards.
```

Argument

`(rest): true`   `bool`

Whether or not the function should return the rest of the cards after splitting. If `true`, the returned array will contain all cards in the input deck, with the remaining cards (i.e. the non-split cards) at the end. Default is `true`. If `false`, the returned array will contain only the split cards, and those which are not contained in the split size will be discarded.

Argument

`(rng): auto`   `(suiji.rng)`

The **random number generator** (rng) to use, from the SUIJI[12] package.

- **Default value** is `auto`, which corresponds to creating a new `rng` internally and automatically, using a seed derived from the cards. *(This means that the split will be deterministic based on the input cards, and the same cards will always produce the same groups.)* Using the `auto` value is recommended for most use cases, especially when this is the only time you need a pseudo-random number generator in your code.
- The other option is to provide an **externally-defined** `rng`, which will be used and updated according to the Suiji library's conventions. This is useful when you want to use the same `rng` across multiple calls, or when you want to use a specific random number generator that you have already created.

The value of this parameter also influences the return value of the function:
- If `rng` is set to `auto`, the function will return **only the produced groups**, i.e. an array of arrays of cards.
- If `rng` is set to a specific `rng`, the function will return **a tuple** containing the updated `rng` and the produced content.

---

[12]https://typst.app/universe/package/suiji

II.5 Arranging hands

This section covers the **sorting and organizing functions** available in the `arr` module in the DECKZ package. It explains how to sort cards by rank, suit, and other criteria, providing a foundation for building more complex card games and applications.

```
#deckz.arr.count-ranks      #deckz.arr.group-ranks      #deckz.arr.sort
#deckz.arr.count-suits      #deckz.arr.group-suits      #deckz.arr.sort-by-order
#deckz.arr.find-ranks       #deckz.arr.order-comparator  #deckz.arr.sort-by-score
#deckz.arr.find-suits       #deckz.arr.score-comparator
```

#deckz.arr.count-ranks(⟨cards⟩, ⟨add-zero⟩: false, ⟨allow-invalid⟩: false) → `dictionary`

Get the count of each rank in the given cards. This function returns a dictionary where the keys are the ranks and the values are the counts of how many times each rank appears in the provided cards.

> ┌─ Argument ───┐
> `⟨cards⟩`   `array`
>
> The cards to check for rank counts. This can be a list or any iterable collection of card codes.

> ┌─ Argument ───┐
> `⟨add-zero⟩: false`   `bool`
>
> If true, the function will add a zero count for ranks that do not appear in the cards. Default: false

> ┌─ Argument ───┐
> `⟨allow-invalid⟩: false`   `bool`
>
> If true, the function will also count cards that are not valid according to the `extract-card-data` function. This is useful for debugging or when you want to include all cards regardless of their validity Default: false

#deckz.arr.count-suits(⟨cards⟩, ⟨add-zero⟩: false, ⟨allow-invalid⟩: false) → `dictionary`

Get the count of each suit in the given cards. This function returns a dictionary where the keys are the suits and the values are the counts of how many times each suit appears in the provided cards.

> ┌─ Argument ───┐
> `⟨cards⟩`   `array`
>
> The cards to check for suit counts. This can be a list or any iterable collection of card codes.

┌─ Argument ──┐
│ `(add-zero): false`   `bool` │
│   │
│ If true, the function will add a zero count for suits that do not appear in the cards. │
│ Default: false   │
└───┘

┌─ Argument ──┐
│ `(allow-invalid): false`   `bool` │
│   │
│ If true, the function will also count cards that are not valid according to the `extract-` │
│ `card-data` function. This is useful for debugging or when you want to include all cards │
│ regardless of their validity Default: false   │
└───┘

`#deckz.arr.find-ranks((cards), (allow-invalid): false)` → `dictionary`

For each rank, check if it is present in the given cards. This function returns a dictionary where the keys are the ranks and the values are booleans indicating whether that rank is present in the provided cards. This is more efficient than counting the ranks with the function `#deckz.arr.count-ranks`, as it only checks for presence and does not count occurrences.

┌─ Argument ──┐
│ `(cards)`   `array` │
│   │
│ The cards to check for rank presence. This can be a list or any iterable collection of │
│ card codes.   │
└───┘

┌─ Argument ──┐
│ `(allow-invalid): false`   `bool` │
│   │
│ If true, the function will also count cards that are not valid according to the `extract-` │
│ `card-data` function. This will add one entry with the key `none` to the result if there │
│ are invalid cards. If false, the function will panic if it finds an invalid card, returning │
│ none. Default: false   │
└───┘

`#deckz.arr.find-suits((cards), (allow-invalid): false)` → `dictionary`

For each suit, check if it is present in the given cards. This function returns a dictionary where the keys are the suits and the values are booleans indicating whether that suit is present in the provided cards. This is more efficient than counting the suits with the function `#deckz.arr.count-suits`, as it only checks for presence and does not count occurrences.

┌─ Argument ──┐
│ `(cards)`   `array` │
│   │
│ The cards to check for suit presence. This can be a list or any iterable collection of │
│ card codes.   │
└───┘

┌─ Argument ──┐
│ `(allow-invalid): false`   `bool` │
│   │
│ If true, the function will also count cards that are not valid (i.e. whose rank symbol │
│ does not correspond to any of the known ranks). This will add one entry with the key │
└───┘

> none to the result if there are invalid cards. If false, the function will panic if it finds an invalid card, returning none. Default: false

#`deckz.arr`.`group-ranks`(`(cards)`, `(add-zero)`: `false`, `(allow-invalid)`: `false`) → `dictionary`

Sort the given cards by their rank. This function returns a dictionary where the keys are the ranks and the values are arrays of cards that have that rank.

> ─ Argument ──────────────────────────────────
> `(cards)`   `array`
>
> The cards to sort by rank. This can be a list or any iterable collection of card codes.

> ─ Argument ──────────────────────────────────
> `(add-zero)`: `false`   `bool`
>
> If true, the function will add an entry for each rank, even if the rank does not appear in the cards. This entry will have the rank as the key and an empty array as the value. Default: false

> ─ Argument ──────────────────────────────────
> `(allow-invalid)`: `false`   `bool`
>
> If true, the function will also count cards that are not valid according to the `extract-card-data` function. If false, the function will simply ignore invalid cards. (No error will be raised.) Default: false

#`deckz.arr`.`group-suits`(`(cards)`, `(add-zero)`: `false`, `(allow-invalid)`: `false`) → `dictionary`

Sort the given cards by their suit. This function returns a dictionary where the keys are the suits and the values are arrays of cards that have that suit.

> ─ Argument ──────────────────────────────────
> `(cards)`   `array`
>
> The cards to sort by suit. This can be a list or any iterable collection of card codes.

> ─ Argument ──────────────────────────────────
> `(add-zero)`: `false`   `bool`
>
> If true, the function will add an entry for each suit, even if the suit does not appear in the cards. This entry will have the suit as the key and an empty array as the value. Default: false

> ─ Argument ──────────────────────────────────
> `(allow-invalid)`: `false`   `bool`

> If true, the function will also count cards that are not valid according to the `extract-card-data` function. If false, the function will simply ignore invalid cards. (No error will be raised.) Default: false

#`deckz.arr`.`order-comparator`(⟨card-it⟩) → `(key, key)`

Compare two cards based on their order in a standard sorted deck. The order compares two cards by their suit (first) and rank (second). Suits and ranks are ordered according to their attributes "suit-order" and "rank-order". These are defined by default as:
- Suits: Hearts, Diamonds, Clubs, Spades
- Ranks: A, 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K

> ┌─ Argument ───┐
> `⟨card-it⟩`   `card`
>
>   The iterator of the card, from which the suit and rank are extracted.
> └───┘

#`deckz.arr`.`score-comparator`(⟨card-it⟩) → `key`

Compare two cards based on their score, given the card data. The score is determined by the rank of the card, where Ace comes first, followed by King, Queen, Jack, and then numbered cards from 10 to 2: A, K, Q, J, 10, 9, 8, 7, 6, 5, 4, 3, 2.

> ┌─ Argument ───┐
> `⟨card-it⟩`   `card`
>
>   The iterator of the card, from which the rank is extracted.
> └───┘

#`deckz.arr`.`sort`(⟨cards⟩, ⟨by⟩: `auto`) → `array`

Sort a list of cards based on a specified key. The key can be "score", "order", or any other attribute of the card. If the key is **"score"**, the cards are sorted by their score (see #`deckz.arr`.`sort-by-score`). If the key is **"order"**, the cards are sorted by their order in a standard sorted deck (see #`deckz.arr`.`sort-by-order`). If the key is not specified or is `auto`, the behavior defaults to sorting by **order**. Other keys will be interpreted as attributes of the card and sorted accordingly, as if using the `sorted` method with that key.

> ┌─ Argument ───┐
> `⟨cards⟩`   `array`
>
>   The cards to sort.
> └───┘

> ┌─ Argument ───┐
> `⟨by⟩`: `auto`   `str` │ `auto`
>
>   The key to sort by. Can be "score", "order", or any other attribute of the card. Defaults to "order" if not specified.
> └───┘

#`deckz.arr`.`sort-by-order`(⟨cards⟩) → `cards`

Sort a list of cards by their order in a standard sorted deck. The cards are sorted in ascending order based on their suit and rank.

```
┌─ Argument ─────────────────────────────────────────────────────────┐
│  (cards)                                                     array   │
│                                                                     │
│     The cards to sort.                                              │
└─────────────────────────────────────────────────────────────────────┘
```

#deckz.arr.sort-by-score((cards)) → `cards`

Sort a list of cards by their score. The cards are sorted in descending order based on their score, which is determined by the rank of the card. The order is as follows: Ace, King, Queen, Jack, 10, 9, 8, 7, 6, 5, 4, 3, 2.

```
┌─ Argument ─────────────────────────────────────────────────────────┐
│  (cards)                                                     array   │
│                                                                     │
│     The cards to sort.                                              │
└─────────────────────────────────────────────────────────────────────┘
```

II.6 Evaluating hands

This section provides an overview of the **scoring functions** from the `val` module of the DECKZ package, which are essential for evaluating hands in card games. Such functions are helpful for assessing the value of a hand based on various criteria and combinations, such as n-of-a-kind, flushes, and straights.

```
#deckz.val.extract              #deckz.val.has-five-of-a-kind  #deckz.val.is-flush
#deckz.val.extract-five-of-a-   #deckz.val.has-flush            #deckz.val.is-four-of-a-kind
kind                            #deckz.val.has-four-of-a-kind   #deckz.val.is-full-house
#deckz.val.extract-flush        #deckz.val.has-full-house       #deckz.val.is-high-card
#deckz.val.extract-four-of-a-   #deckz.val.has-high-card        #deckz.val.is-n-of-a-kind
kind                            #deckz.val.has-n-of-a-kind      #deckz.val.is-pair
#deckz.val.extract-high-card    #deckz.val.has-pair             #deckz.val.is-straight
#deckz.val.extract-highest      #deckz.val.has-straight         #deckz.val.is-straight-flush
#deckz.val.extract-n-of-a-      #deckz.val.has-straight-flush   #deckz.val.is-three-of-a-kind
kind                            #deckz.val.has-three-of-a-      #deckz.val.is-two-pairs
#deckz.val.extract-pair         kind
#deckz.val.extract-straight     #deckz.val.has-two-pairs
#deckz.val.extract-straight-    #deckz.val.is-five-of-a-kind
flush
#deckz.val.extract-three-of-
a-kind
#deckz.val.extract-two-pairs
```

#deckz.val.extract(`(scoring-combination)`, `(cards)`) → `array`

Extract a scoring combination from the given cards. This function accepts a scoring combination name as a string and returns the corresponding hand. The scoring combinations are defined as follows:

- "high-card": A single card with a valid rank.
- "pair": Two cards of the same rank.
- "two-pairs": Two distinct pairs of cards, each of the same rank.
- "three-of-a-kind": Three cards of the same rank.
- "straight": Five consecutive ranks, regardless of suit.
- "flush": Five cards of the same suit.
- "full-house": Three of a kind and a pair.
- "four-of-a-kind": Four cards of the same rank.
- "straight-flush": A straight and a flush at the same time.
- "five-of-a-kind": Five cards of the same rank.

If the scoring combination is not recognized, the function will panic. If the scoring combination is recognized, but the cards don't have such combination, the function will return an empty array.

> ┌─ Argument ───┐
> `(scoring-combination)`   `str`
>
> The scoring combination to extract.
> └──┘

#`deckz.val`.`extract-five-of-a-kind`(⟨cards⟩) → `array`

Get all five of a kind from the given cards. This function returns an array of arrays, where each inner array is a **hand** that contains five cards of the same rank. If there are no five of a kind, it returns an empty array.

┌─ Argument ───┐
│ ⟨cards⟩   `array` │
│   │
│   The cards to check for five of a kind. This can be an array or any iterable collection of │
│   card codes.   │
└──┘

#`deckz.val`.`extract-flush`(⟨cards⟩, ⟨n⟩: 5) → `array`

Get all flushes from the given cards. This function returns an array of arrays, where each inner array is a **hand** that contains n cards of the same suit. If there are no flushes, it returns an empty array.

┌─ Argument ───┐
│ ⟨cards⟩   `array` │
│   │
│   The cards to check for a flush. This can be an array or any iterable collection of card │
│   codes.   │
└──┘

┌─ Argument ───┐
│ ⟨n⟩: 5   `int`   │
│   │
│   The number of cards required for a flush. Default: 5   │
└──┘

#`deckz.val`.`extract-four-of-a-kind`(⟨cards⟩) → `array`

Get all four of a kind from the given cards. This function returns an array of arrays, where each inner array is a **hand** that contains four cards of the same rank. If there are no four of a kind, it returns an empty array.

┌─ Argument ───┐
│ ⟨cards⟩   `array` │
│   │
│   The cards to check for four of a kind. This can be an array or any iterable collection │
│   of card codes.   │
└──┘

#`deckz.val`.`extract-high-card`(⟨cards⟩) → `array`

Get all "high card" hands from the given cards. This function returns an array of arrays, where each inner array is a **hand** that contains a single card with a valid rank. If there are no high card hands, it returns an empty array.

> **Argument**
>
> `(cards)`   `array`
>
> The cards to check for high card. This can be a list or any iterable collection of card codes.

#`deckz.val`.`extract-highest`(`(cards)`, `(sort)`: `true`) → `array`

Return all the combinations of the first valid scoring combination found in the given cards, starting from the highest scoring combination. This function checks the cards for the highest scoring combination and returns the corresponding hand. If the cards do not contain any valid scoring combination, the function will return an empty array.

> **Argument**
>
> `(cards)`   `array`
>
> The cards to check for the highest scoring combination. This can be an array or any iterable collection of card codes.

> **Argument**
>
> `(sort)`: `true`
>
> If true, the function will first sort the cards by their score. This is needed if you want the highest combination among multiple combinations of the same type. E.g. if you have more than one pair, the function will return the pair with the highest rank (i.e. the card with the highest score). Default: true

#`deckz.val`.`extract-n-of-a-kind`(`(cards)`, `(n)`: `2`) → `array`

Returns all possible n-of-a-kind hands from the given cards. This function returns an array of arrays, where each inner array is a **hand** that contains n cards of the same rank. Hands are sorted by the rank of the cards, according to the order defined in the `ranks` module. If there are no n-of-a-kind hands, it returns an empty array.

> **Argument**
>
> `(cards)`   `array`
>
> The cards from which we try to extract a n-of-a-kind.

> **Argument**
>
> `(n)`: `2`
>
> The number of cards that must have the same rank to count as n-of-a-kind. Default: 2

#`deckz.val`.`extract-pair`(`(cards)`) → `array`

Get all pairs from the given cards. This function returns an array of arrays, where each inner array is a **hand** that contains two cards of the same rank. If there are no pairs, it returns an empty array.

> **Argument**
>
> `(cards)`   `array`

> The cards to check for pairs. This can be an array or any iterable collection of card codes.

#`deckz.val`.`extract-straight`(⟨cards⟩, ⟨n⟩: 5) → `array`

Get all straights from the given cards. This function returns an array of strings, where each string is a **hand** that contains n consecutive ranks. If there are no straights, it returns an empty array.

> ┌─ Argument ───┐
> ⟨cards⟩   `array`
>
> The cards to check for a straight. This can be an array or any iterable collection of card codes.
> └──┘

> ┌─ Argument ───┐
> ⟨n⟩: 5   `int`
>
> The number of consecutive ranks required for a straight. Default: 5
> └──┘

#`deckz.val`.`extract-straight-flush`(⟨cards⟩, ⟨n⟩: 5) → `array`

Get all straight flushes from the given cards. This function returns an array of arrays, where each inner array is a **hand** that contains n consecutive ranks of the same suit. If there are no straight flushes, it returns an empty array.

> ┌─ Argument ───┐
> ⟨cards⟩   `array`
>
> The cards to check for a straight flush. This can be an array or any iterable collection of card codes.
> └──┘

> ┌─ Argument ───┐
> ⟨n⟩: 5   `int`
>
> The number of cards required for a straight flush. Default: 5
> └──┘

#`deckz.val`.`extract-three-of-a-kind`(⟨cards⟩) → `array`

Get all three of a kind from the given cards. This function returns an array of arrays, where each inner array is a **hand** that contains three cards of the same rank. If there are no three of a kind, it returns an empty array.

> ┌─ Argument ───┐
> ⟨cards⟩   `array`
>
> The cards to check for three of a kind. This can be an array or any iterable collection of card codes.
> └──┘

#`deckz.val`.`extract-two-pairs`(⟨cards⟩) → `array`

Get all two pairs from the given cards. This function returns an array of arrays, where each inner array is a **hand** that contains two distinct pairs of cards, each of the same rank. If there are no two pairs, it returns an empty array.

> **Argument**
> `(cards)`   `array`
>
> The cards to check for two pairs. This can be an array or any iterable collection of card codes.

#deckz.val.has-five-of-a-kind((cards)) → `bool`

Check if the given cards contain five of a kind. Five of a kind is defined as five cards of the same rank.

> **Argument**
> `(cards)`   `array`
>
> The cards to check for five of a kind. This can be an array or any iterable collection of card codes.

#deckz.val.has-flush((cards), (n): 5) → `bool`

Check if the given cards contain a flush. A flush is defined as having at least `n` cards of the same suit.

> **Argument**
> `(cards)`   `array`
>
> The cards to check for a flush. This can be an array or any iterable collection of card codes.

> **Argument**
> `(n): 5`   `int`
>
> The number of cards required for a flush. Default: 5

#deckz.val.has-four-of-a-kind((cards)) → `bool`

Check if the given cards contain four of a kind. Four of a kind is defined as four cards of the same rank.

> **Argument**
> `(cards)`   `array`
>
> The cards to check for four of a kind. This can be an array or any iterable collection of card codes.

#deckz.val.has-full-house((cards)) → `bool`

Check if the given cards contain a full house. A full house is defined as having three of a kind and a pair.

> **Argument**
> `(cards)`   `array`

> The cards to check for a full house. This can be an array or any iterable collection of card codes.

#deckz.val.has-high-card((cards)) → `bool`

Check if the given cards contain a high card. A high card is defined as having at least one card with a valid rank, regardless of which rank it is.

┌─ Argument ──┐
│ (cards)   `array` │
│   │
│ The cards to check for high card. This can be a list or any iterable collection of card │
│ codes.   │
└──┘

#deckz.val.has-n-of-a-kind((cards), (n): 2) → `bool`

Check if the given cards contain n-of-a-kind. n-of-a-kind is defined as having at least n cards of the same rank

┌─ Argument ──┐
│ (cards)   `array` │
│   │
│ The cards to check for n-of-a-kind. This can be a list or any iterable collection of │
│ card codes.   │
└──┘

┌─ Argument ──┐
│ (n): 2   `int` │
│   │
│ The number of cards that must have the same rank to count as n-of-a-kind. Default: 2 │
└──┘

#deckz.val.has-pair((cards)) → `bool`

Check if the given cards contain a pair. A pair is defined as two cards of the same rank.

┌─ Argument ──┐
│ (cards)   `array` │
│   │
│ The cards to check for a pair. This can be an array or any iterable collection of card │
│ codes.   │
└──┘

#deckz.val.has-straight((cards), (n): 5) → `bool`

Check if the given cards contain a straight. A straight is defined as five consecutive ranks, regardless of suit.

┌─ Argument ──┐
│ (cards)   `array` │
│   │
│ The cards to check for a straight. This can be an array or any iterable collection of │
│ card codes.   │
└──┘

┌─ Argument ──┐
│ (n): 5   `int` │
│   │
└──┘

> The number of consecutive ranks required for a straight. Default: 5

`#deckz.val.has-straight-flush((cards), (n): 5)` → `bool`

Check if the given cards contain a straight flush. A straight flush is defined as having a straight and a flush at the same time.

Note: This function may accept more than `n` cards (default: more than 5 cards). This means that it will return true if there is a straight and if there is a flush, regardless of the number of cards. This means that the two conditions may not be met by the same cards.

> — Argument —————————————————————————————
> `(cards)`   `array`
>
> The cards to check for a straight flush. This can be an array or any iterable collection of card codes.

> — Argument —————————————————————————————
> `(n): 5`   `int`
>
> The number of cards required for a straight flush. Default: 5

`#deckz.val.has-three-of-a-kind((cards))` → `bool`

Check if the given cards contain three of a kind. Three of a kind is defined as three cards of the same rank.

> — Argument —————————————————————————————
> `(cards)`   `array`
>
> The cards to check for three of a kind. This can be an array or any iterable collection of card codes.

`#deckz.val.has-two-pairs((cards))` → `bool`

Check if the given cards contain two pairs. Two pairs are defined as two distinct pairs of cards, each of the same rank.

> — Argument —————————————————————————————
> `(cards)`   `array`
>
> The cards to check for two pairs. This can be an array or any iterable collection of card codes.

`#deckz.val.is-five-of-a-kind((cards))` → `bool`

Check if the given cards correspond to five of a kind, i.e. if they are five cards of the same rank.

> — Argument —————————————————————————————
> `(cards)`   `array`
>
> The cards to check for five of a kind. This can be an array or any iterable collection of card codes.

`#deckz.val.is-flush((cards), (n): 5)` → `bool`

Check if the given cards correspond to a flush, i.e. the hand is composed by n cards of the same suit. This is done by checking that there are n cards in total, and that there is only one suit present in the hand.

> ── Argument ──
> `(cards)` `array`
>
> The cards to check for a flush. This can be an array or any iterable collection of card codes.

> ── Argument ──
> `(n): 5` `int`
>
> The number of cards required for a flush. Default: 5

#deckz.val.is-four-of-a-kind((cards)) → bool

Check if the given cards correspond to four of a kind, i.e. if they are four cards of the same rank.

> ── Argument ──
> `(cards)` `array`
>
> The cards to check for four of a kind. This can be an array or any iterable collection of card codes.

#deckz.val.is-full-house((cards)) → bool

Check if the given cards correspond to a full house. A full house is defined as having three of a kind and a pair.

> ── Argument ──
> `(cards)` `array`
>
> The cards to check for a full house. This can be an array or any iterable collection of card codes.

#deckz.val.is-high-card((cards)) → bool

Check if the given cards correspond to a high card. A high card is defined as a single card with a valid rank.

> ── Argument ──
> `(cards)` `array`
>
> The cards to check for high card. This can be a list or any iterable collection of card codes.

#deckz.val.is-n-of-a-kind((cards), (n): 2) → bool

Check if the given cards correspond to a "n-of-a-kind", i.e. if they are n cards of the same rank. This is a stricter version of the `has-n-of-a-kind` function, as it checks that all cards correspond to the requested hand.

> ── Argument ──
> `(cards)` `array`

> The cards to check for n-of-a-kind. This can be a list or any iterable collection of card codes.

> **Argument**
>
> `⟨n⟩: 2`   `int`
>
> The number of cards that must have the same rank to count as n-of-a-kind. Default: 2

`#deckz.val`.`is-pair`(`⟨cards⟩`) → `bool`

Check if the given cards correspond to a pair, i.e. if they are two cards of the same rank.

> **Argument**
>
> `⟨cards⟩`   `array`
>
> The cards to check for a pair. This can be an array or any iterable collection of card codes.

`#deckz.val`.`is-straight`(`⟨cards⟩`, `⟨n⟩: 5`)

Check if the given cards correspond to a straight. A straight is defined as having exactly n consecutive ranks, regardless of suit.

> **Argument**
>
> `⟨cards⟩`   `array`
>
> The cards to check for a straight. This can be an array or any iterable collection of card codes.

> **Argument**
>
> `⟨n⟩: 5`   `int`
>
> The number of consecutive ranks required for a straight. Default: 5

`#deckz.val`.`is-straight-flush`(`⟨cards⟩`, `⟨n⟩: 5`) → `bool`

Check if the given cards correspond to a straight flush, i.e. a straight and a flush at the same time.

> **Argument**
>
> `⟨cards⟩`   `array`
>
> The cards to check for a straight flush. This can be an array or any iterable collection of card codes.

> **Argument**
>
> `⟨n⟩: 5`   `int`
>
> The number of cards required for a straight flush. Default: 5

`#deckz.val`.`is-three-of-a-kind`(`⟨cards⟩`) → `bool`

Check if the given cards correspond to three of a kind, i.e. if they are three cards of the same rank.

> **Argument**
>
> ```
> (cards)                                                                    array
> ```
>
> The cards to check for three of a kind. This can be an array or any iterable collection
> of card codes.

#`deckz.val`.`is-two-pairs`(`(cards)`) → `bool`

Check if the given cards correspond to two pairs, i.e. if they are four cards with two distinct
pairs of the same rank. This is done by checking that there are four cards in total, and that
there are two distinct pairs of the same rank.

> **Argument**
>
> ```
> (cards)                                                                    array
> ```
>
> The cards to check for two pairs. This can be an array or any iterable collection of
> card codes.

II.7 Language-aware card symbols

DECKZ automatically adapts the rendering of card rank symbols based on the document's language. This process is seamless: users only need to set the desired language using the text command, and DECKZ will adjust the symbols accordingly. No additional configuration is required.

The current version of DECKZ supports the following **languages**:

EN	A Ace	2 Two	3 Three	4 Four	5 Five	6 Six	7 Seven	8 Eight	9 Nine	10 Ten	J Jack	Q Queen	K King
IT	A Asso	2 Due	3 Tre	4 Quattro	5 Cinque	6 Sei	7 Sette	8 Otto	9 Nove	10 Dieci	J Fante	Q Regina	K Re
FR	A As	2 Deux	3 Trois	4 Quatre	5 Cinq	6 Six	7 Sept	8 Huit	9 Neuf	10 Dix	V Valet	D Dame	R Roi
DE	A Ass	2 Zwei	3 Drei	4 Vier	5 Fünf	6 Sechs	7 Sieben	8 Acht	9 Neun	10 Zehn	B Bube	D Dame	K König
RU	A Туз	2 Двойка	3 Тройка	4 Четвёр- ка	5 Пятёр- ка	6 Шестёр- ка	7 Семёр- ка	8 Восьь- мёрка	9 Девят- ка	10 Десят- ка	В Валет	Д Дама	К Король

The localization is powered by the LINGUIFY[13] package, thanks to the generous contribution of the users who helped translating card names and symbols.

```
#let seq = ("10C", "JH", "QS", "KD",
"AC")

#set text(lang: "en")
#deckz.line(..seq, format: "small")

#set text(lang: "it")
#deckz.line(..seq, format: "small")

#set text(lang: "fr")
#deckz.line(..seq, format: "small")
```

Part III
Examples

The following examples showcase more advanced and interesting **use cases** of DECKZ. In this Section, you can find how DECKZ can be used to represent real game states, compare card formats, and display entire decks in creative ways.

III.1 Displaying the current state of a game

You can use DECKZ to display the **current state of a game**, such as the cards in hand, the cards on the table, and the deck.

```
#let player-mat(body) = box(
  stroke: olive.darken(20%),
  fill: olive.lighten(10%),
  radius: (top: 50%, bottom: 5%),
  inset: 15%,
  body
)

#text(white, font: "Roboto Slab", weight: "semibold")[
  #box(fill: olive,
    width: 100%, height: 12cm,
    inset: 4mm, radius: 2mm
  )[

    #place(center + bottom)[
      #player-mat[
       #deckz.hand(format: "small", width: 3cm, "9S", "10H", "4C", "4D", "2D")
        Alice
      ]
    ]
    #place(left + horizon)[
      #rotate(90deg, reflow: true)[
        #player-mat[
         #deckz.hand(format: "small", width: 3cm, "AS", "JH", "JC", "JD", "3D")
          #align(center)[Bob]
        ]
      ]
    ]
    #place(center + top)[
      #rotate(180deg, reflow: true)[
        #player-mat[
         #deckz.hand(format: "small", width: 3cm, "KH", "8H", "7H", "5C", "3C")
          #rotate(180deg)[Carol]
        ]
      ]
    ]
```

```
    ]
    #place(right + horizon)[
      #rotate(-90deg, reflow: true)[
        #player-mat[
         #deckz.hand(format: "small", width: 3cm, "6S", "3H", "2H", "QC", "9C")
          #align(center)[Dave]
        ]
      ]
    ]
    #place(center + horizon)[
      #deckz.deck(format: "small", angle: 80deg, height: 8mm, "back")
    ]
  ]
]
```

In this situation, Alice has a ***Pair of Four*** (#deckz.inline("4C"),
#deckz.inline("4D")). _*What should the player do?*_
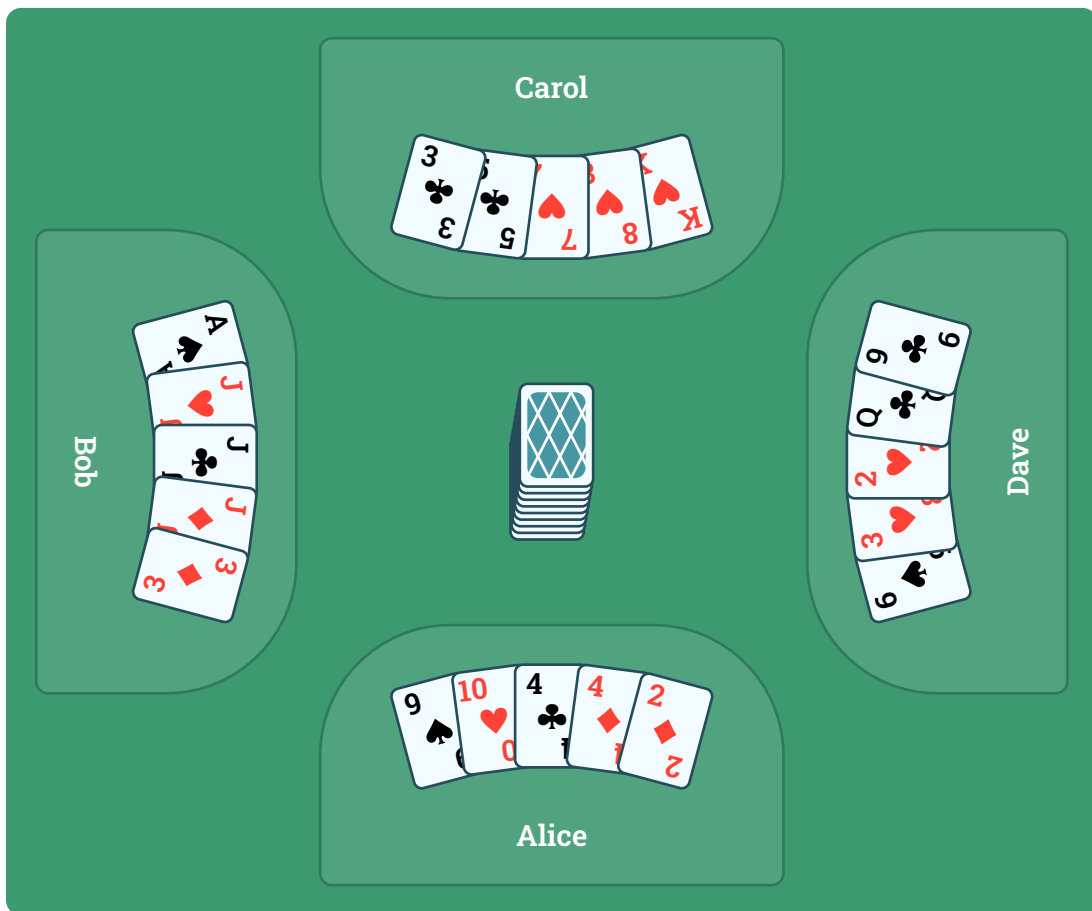
In this situation, Alice has a **Pair of Four** (4♣ , 4♦). *What should the player do?*

III.2 Comparing different formats

You can use DECKZ to **compare different formats** of the same card, or to show how a card looks in different contexts.

```
#let cards-window(body) = block(fill: white, width: 100%, inset: 5pt, body)

#block(fill: gray.lighten(60%), inset: 5pt, breakable: false)[

  #let example-cards = ("AS", "5H", "10C", "QD")

  #text(blue)[`inline`] --- A minimal format where the rank and suit are
displayed directly within the flow of text; perfect for quick references.
  #cards-window[
    #deckz.line(..example-cards, format: "inline", width: 100%)
  ]

  #text(blue)[`mini`] --- The smallest visual format: a compact rectangle
showing the rank at the top and the suit at the bottom.
  #cards-window[
    #deckz.line(..example-cards, format: "mini", width: 100%)
  ]

  #text(blue)[`small`] --- A slightly larger card with rank indicators on
opposite corners and a central suit symbol; ideal for tight layouts with
better readability.
  #cards-window[
    #deckz.line(..example-cards, format: "small", width: 100%)
  ]

  #text(blue)[`medium`] --- A fully structured card layout featuring proper suit
placement and figures. Rank and suit appear in two opposite corners, offering
a realistic visual.
  #cards-window[
    #deckz.line(..example-cards, format: "medium", width: 100%)
  ]

  #text(blue)[`large`] --- The most detailed format, with corner summaries on all
four corners and an expanded layout; great for presentations or printable decks.
  #cards-window[
    #deckz.line(..example-cards, format: "large", width: 100%)
  ]
]
```

`inline` — A minimal format where the rank and suit are displayed directly within the flow of text; perfect for quick references.

A♠   5♥   10♣   Q♦

`mini` — The smallest visual format: a compact rectangle showing the rank at the top and the suit at the bottom.

| A | 5 | 10 | Q |
| ♠ | ♥ | ♣ | ♦ |

`small` — A slightly larger card with rank indicators on opposite corners and a central suit symbol; ideal for tight layouts with better readability.

`medium` — A fully structured card layout featuring proper suit placement and figures. Rank and suit appear in two opposite corners, offering a realistic visual.
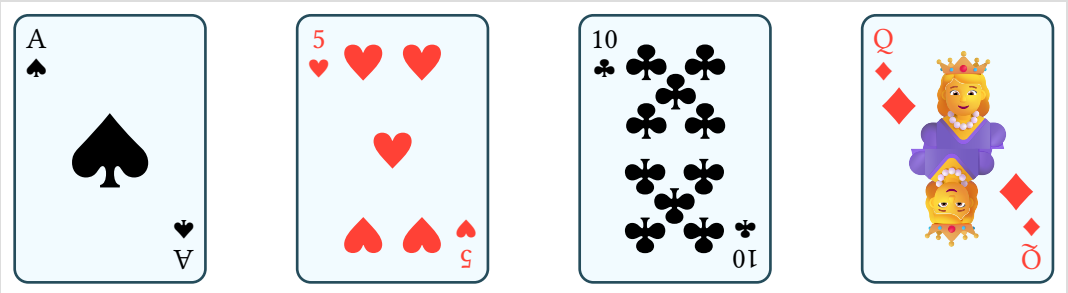
`large` — The most detailed format, with corner summaries on all four corners and an expanded layout; great for presentations or printable decks.
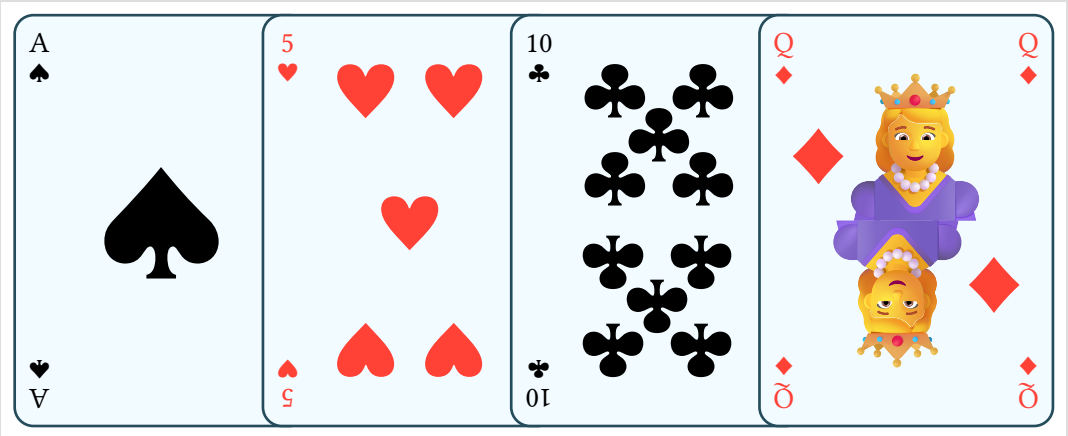
III.3 Displaying a full deck

You can use DECKZ to display a **full deck of cards**, simply by retrieving the deckz.deck52 array, which contains all 52 standard playing cards.

```
#text(white, font: "Oldenburg", size: 10pt)[
  #block(fill: aqua.darken(40%), inset: 4mm, radius: 2mm)[
    #deckz.hand(
      angle: 270deg,
      width: 5.8cm,
      format: "large",
      noise: 0.05,
      ..deckz.deck52
    )
    #place(center + horizon)[#text(size: 20pt, baseline: 8pt)[Deckz]]
  ]
]
```

III.4 Randomized game with card scoring

You can use DECKZ to create a **randomized *Texas Hold'em*-like game**, where players are dealt random hands and their best hands are determined based on the cards on the table.

```
// Defining players and their hands
#let players = ("Alice", "Bob", "Carol", "David")
#let (players-hands, board-cards) = deckz.mix.split(
  deckz.deck52, // Start with a standard deck of 52 cards
  size: ((players.len(), 2), 5), // 2 cards per player + 5 board cards
  rest: false,
)

#set align(center)
#set text(fill: white, size: 13pt, font: "Arvo")

#block(
  fill: olive,
  inset: 15pt,
  radius: 5pt,
  breakable: false,
)[
  // Board cards
  *Board*
  #deckz.hand(
    format: "small",
    width: 4cm,
    angle: 20deg,
    noise: 0.2,
    ..board-cards
  )
  // Displaying the deck of cards
  #place(right + top)[
    #deckz.deck(
      format: "small",
      angle: 90deg,
      noise: 0.2,
      "back"
    )
  ]

  #v(1cm)
  #set table.cell(
    fill: olive.darken(20%),
    stroke: olive.darken(10%) + 2pt,
  )

  // Players' hands
  #table(
    columns: players.len(),
```

```
      column-gutter: 1fr,

    ..players-hands
      .zip(players)
      .map(((hand, player)) => [
        #player
        #deckz.hand(
          format: "small",
          width: 1cm,
          angle: 30deg,
          noise: 0.4,
          ..hand
        )
      ])
  )
]

#set align(left)
#set text(fill: black, size: 12pt)

Which is the *best hand* each player can make?
#for (player, hand) in players.zip(players-hands) [

  - *#player*
  #box(
    fill: olive,
    inset: 8pt,
    radius: 15%,
  )[
    #(hand + board-cards).map(deckz.mini).join("  ")
  ]
  $stretch(=>)^" Best hand "$
  #let best = deckz.val.extract-highest(hand + board-cards, sort: true).first()
  #box(
    fill: none,
    stroke: (paint: green, thickness: 3pt, dash: "dashed"),
    inset: 8pt,
    radius: 45%,
  )[
    #best.map(deckz.mini).join("  ")
  ]
]
```
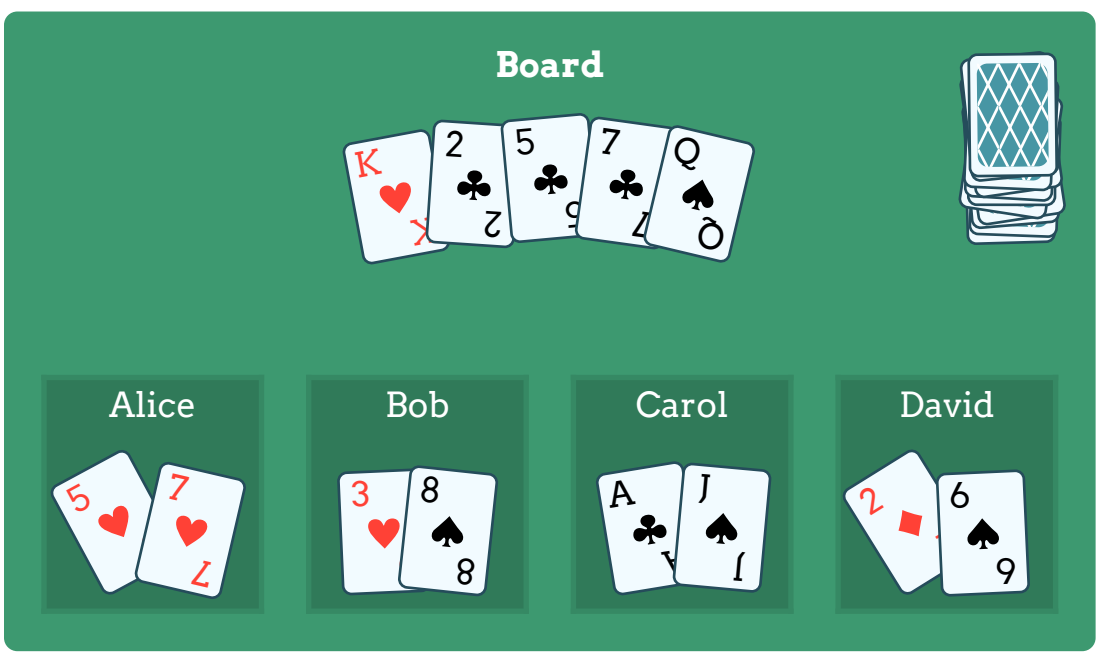
Board

| K♥ | 2♣ | 5♣ | 7♣ | Q♠ |

| Alice | Bob | Carol | David |
| 5♥ 7♥ | 3♥ 8♠ | A♣ J♠ | 2♦ 6♠ |

Which is the **best hand** each player can make?

- **Alice**

| 5♥ | 7♥ | K♥ | 2♣ | 5♣ | 7♣ | Q♠ |

Best hand ⟶ | 7♥ | 7♣ | 5♥ | 5♣ |

- **Bob**

| 3♥ | 8♠ | K♥ | 2♣ | 5♣ | 7♣ | Q♠ |

Best hand ⟶ | K♥ |

- **Carol**

| A♣ | J♠ | K♥ | 2♣ | 5♣ | 7♣ | Q♠ |

Best hand ⟶ | A♣ |

- **David**

| 2♦ | 6♠ | K♥ | 2♣ | 5♣ | 7♣ | Q♠ |

Best hand ⟶ | 2♦ | 2♣ |

III.5 Scoring hands

You can use DECKZ to **score hands** in a game, such as Poker, by extracting combinations of cards that form specific hands, such as pairs, three-of-a-kinds, straights, flushes, and so on.

```
// Define and sort the initial hand
#let my-hand = deckz.arr.sort-by-score(("8S", "9S", "10S", "JS", "QS", "9H",
"9D", "9C", "QD", "3D", "4D", "5D"))

#set text(font: "Roboto Slab")
My current *hand*:

#deckz.hand(..my-hand,
  format: "small",
 angle: 10deg,
  width: 12cm,
)

#let show-combination(combination) = [

 Which *#combination* can I make?

  #let combinations = deckz.val.extract(combination, my-hand)
  #if combinations.len() == 0 {
    [None.]
  } else {
   for combo in combinations {
      box(
       fill: eastern,
       inset: 6pt,
       radius: 12pt,
      )[
       #deckz.line(..combo, format: "mini", spacing: 6pt)
    ]
     h(1mm)
   }
  }
]

#show-combination("high-card")
#show-combination("pair")
#show-combination("two-pairs")
#show-combination("three-of-a-kind")
#show-combination("straight")
#show-combination("flush")
#show-combination("full-house")
#show-combination("four-of-a-kind")
#show-combination("straight-flush")
#show-combination("five-of-a-kind")
```

My current **hand**:

Q♠ Q♦ J♠ 10♠ 9♠ 9♥ 9♦ 9♣ 8♠ 5♦ 4♦ 3♦

Which **high-card** can I make?

Q♠ Q♦ J♠ 10♠ 9♠ 9♥ 9♦ 9♣ 8♠ 5♦ 4♦ 3♦

Which **pair** can I make?

Q♠ Q♦   9♠ 9♥   9♠ 9♦   9♠ 9♣   9♥ 9♦   9♥ 9♣   9♦ 9♣

Which **two-pairs** can I make?

Q♠ Q♦ 9♠ 9♥   Q♠ Q♦ 9♠ 9♦   Q♠ Q♦ 9♠ 9♣   Q♠ Q♦ 9♥ 9♦

Q♠ Q♦ 9♥ 9♣   Q♠ Q♦ 9♦ 9♣
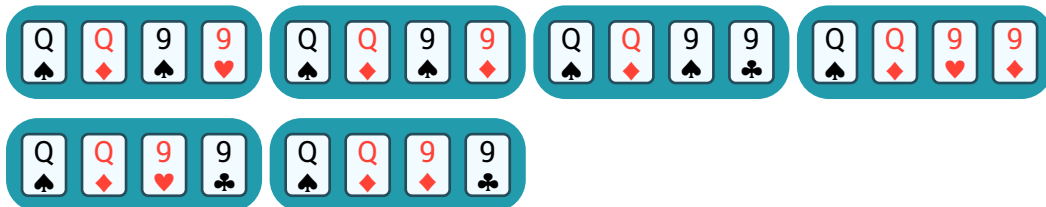
Which **three-of-a-kind** can I make?

9♠ 9♥ 9♦   9♠ 9♥ 9♣   9♠ 9♦ 9♣   9♥ 9♦ 9♣

Which **straight** can I make?

8♠ 9♠ 10♠ J♠ Q♠   8♠ 9♠ 10♠ J♠ Q♦   8♠ 9♥ 10♠ J♠ Q♠

8♠ 9♥ 10♠ J♠ Q♦   8♠ 9♦ 10♠ J♠ Q♠   8♠ 9♦ 10♠ J♠ Q♦

8♠ 9♣ 10♠ J♠ Q♠   8♠ 9♣ 10♠ J♠ Q♦
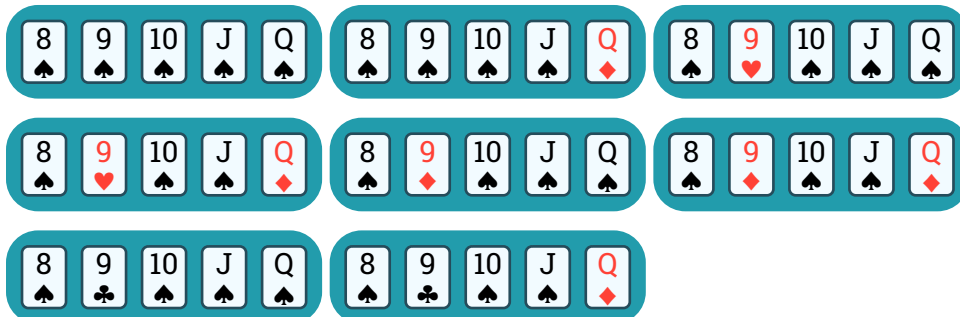
Which **flush** can I make?

Q♠ J♠ 10♠ 9♠ 8♠   Q♦ 9♦ 5♦ 4♦ 3♦

Which **full-house** can I make?

9♠ 9♥ 9♦ Q♠ Q♦   9♠ 9♥ 9♣ Q♠ Q♦   9♠ 9♦ 9♣ Q♠ Q♦

9♥ 9♦ 9♣ Q♠ Q♦
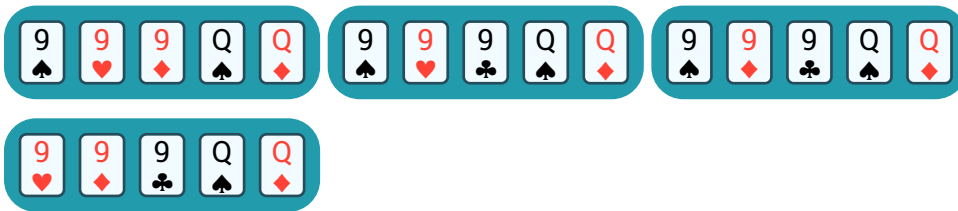
Which **four-of-a-kind** can I make?

9♠ 9♥ 9♦ 9♣
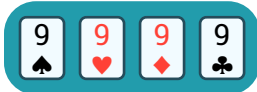
Which **straight-flush** can I make?

8♠ 9♠ 10♠ J♠ Q♠

Which **five-of-a-kind** can I make?

None.

Credits

This package is created by **Michele Dusi**[14] and is licensed under the GNU General Public License v3.0[15].

The **name** is inspired by Typst's drawing package CETZ[16]: it mirrors its sound while hinting at its own purpose: rendering card decks.

All **fonts** used in this package are licensed under the SIL Open Font License, Version 1.1[17] (*Oldenburg*[18], *Arvo*[19]) or the Apache License, Version 2.0[20] (*Roboto Slab*[21]).

The **card designs** are inspired by the standard playing cards, with suit symbols taken from the emoji library of Typst[22].

This project owes a lot to the creators of these **Typst packages**, whose work made DECKZ possible:
- CETZ[23], for handling graphics and for the name inspiration.
- SUIJI[24] and DIGESTIFY[25], for random number generation and hashing respectively.
- LINGUIFY[26], for localization.
- MANTYS[27], TIDY[28], and CODLY[29], for documentation.
- OCTIQUE[30] and SHOWYBOX[31], for documentation styling.

Special thanks to everyone involved in the development of the Typst[32] language and engine, whose efforts made the entire ecosystem possible.

Contributions are welcome!

Found a bug, have an idea, or want to contribute? Feel free to open an **issue** or **pull request** on the *GitHub* repository (micheledusi/Deckz[33]).

Made something cool with Deckz? Let me know — I'd love to feature your work!

---

[14]https://github.com/micheledusi
[15]https://www.gnu.org/licenses/gpl-3.0.en.html
[16]https://typst.app/universe/package/cetz
[17]https://openfontlicense.org
[18]https://fonts.google.com/specimen/Oldenburg
[19]https://fonts.google.com/specimen/Arvo
[20]http://www.apache.org/licenses/
[21]https://fonts.google.com/specimen/Roboto+Slab
[22]https://typst.app/docs/img/reference/symbols/emoji/
[23]https://typst.app/universe/package/cetz
[24]https://typst.app/universe/package/suiji
[25]https://typst.app/universe/package/digestify
[26]https://typst.app/universe/package/linguify
[27]https://typst.app/universe/package/mantys
[28]https://typst.app/universe/package/tidy
[29]https://typst.app/universe/package/codly
[30]https://typst.app/universe/package/octique
[31]https://typst.app/universe/package/showybox
[32]https://typst.app/about/
[33]https://github.com/micheledusi/Deckz

Part IV

Index

M

O

R

S