PnPInk Documentation

None

None

Table of contents

1 PnPInk Documentation

PnPInk is an Inkscape extension suite for data-driven print-and-play production.

Start with Basic Workflow, then use DeckMaker GUI for the current user interface and Export for PDF, PDF/X, and other output formats.

Prefer an offline/manual format? Download the full PDF guide.

2 Introduction

This documentation is also available as a single PDF guide.

2.1 What is PnPInk?

PnPInk is an open-source extension suite for Inkscape that turns it into a practical production environment for print-and-play components: cards, tiles, counters, boards, and player aids.

If you are new to Inkscape: Inkscape is a free, open-source vector editor based on the SVG standard. If you do not have it installed yet, you can download it for Windows, macOS, and Linux from the official website: https://inkscape.org

PnPInk works fully inside Inkscape. You design with normal SVG objects, and PnPInk handles replication, data filling, placement, and pagination automatically.

The output remains editable SVG, and you can export using Inkscape formats such as PDF, PNG, JPG, and SVG.

2.2 The core idea

PnPInk is built around a simple workflow:

1. Draw one component.
2. Describe variations in a dataset.
3. Let PnPInk generate the rest.

You start with a single visual design (a card, a tile, a token face), connect it to a dataset, and PnPInk produces as many instances as you need, placing them on pages and preparing them for printing.

You can begin with defaults. Advanced controls are optional and can be added gradually when you need more precision.

2.3 What you can do immediately

Without advanced syntax, you can already:

- duplicate one template many times,
- change texts and images per instance,
- auto-fill pages,
- generate multi-page output ready to export.

Everything beyond that is incremental.

2.4 First contact: template, IDs, dataset

2.4.1 What is a template in PnPInk?

A template is a normal Inkscape drawing that represents one unit to replicate: one card, one tile, one board section, and so on.

In practice, it is a group of regular SVG objects (text, rects, paths, images, groups), identified by IDs.

PnPInk uses one internal object as template bounding box (`bbox`) to understand:

- template size,
- placement reference,
- replication behavior.

The `bbox` can be a rect or another simple shape. What matters is that its outline correctly wraps the component you want to replicate.

2.4.2 How IDs connect data to graphics

In PnPInk, IDs are the connection between dataset and drawing.

If a dataset column is named `title` and your SVG contains `id="title"`, that object can be updated for each generated row.

Typical usage:

- text IDs: dynamic text replacement,
- rect IDs: image/icon anchors,
- group IDs: visibility or variant control (advanced workflows).

Useful Inkscape panels:

- `Object > Objects...` (`Shift+Ctrl+O`): hierarchy, groups, layers, Z-order, IDs.
- `Object > XML Editor` (`Shift+Ctrl+X`): direct SVG/XML attributes.

2.5 Minimal working example

2.5.1 Dataset notation used in this documentation

To avoid confusion, this guide uses a consistent visual notation:

- header cells: dataset column names (first row),
- regular cells: normal data values,
- first-column cells: cells in column 1 (template/page/layout control).

In dataset tables, headers and first-column cells are also color-highlighted.

2.5.2 Conceptual template structure

Use a basic Inkscape file named `hello_word.svg`.

In Inkscape, the structure can look like this:

```
(g) hello_word_template
 |- (rect) card_bbox        <- template bounding box
 |- (text) title
 |- (text) cost
 |- (rect) art              <- image/icon anchor
 `- (text) text
```

Key points:

- `card_bbox` visually wraps the full component.
- `title` , `cost` , `art` , and `text` are the objects you vary per row.

2.5.3 Dataset example

Spreadsheet-like table view:

card_bbox	title	cost	art	text
`{A4 b=[-5]}.L{p=4x3 g=2}`	Tomatoes	3	tomato	You win 1 tomato
	Mushrooms	5	brown-mushroom	You win 2 mushrooms
	Lemons	2	lemon	Win 1 lemon for every tomato you own

CSV text view (same data):

```
card_bbox,title,cost,art,text
{A4 b=[-5]}.L{p=4x3 g=2},Tomatoes,3,tomato,You win 1 tomato
,Mushrooms,5,brown-mushroom,You win 2 mushrooms
,Lemons,2,lemon,Win 1 lemon for every tomato you own
```

Interpretation:

- The header of the first column is `card_bbox`.
- The first-column cell `{A4 b=[-5]}.L{p=4x3 g=2}` sets page/layout context for this dataset block.
- Empty first-column cells continue using the same template/page context.
- Each row generates one component instance.
- `title` and `cost` update text fields.
- `art` provides the image source for the `art` anchor.
- `text` updates the text object with `id="text"`.

With defaults only, PnPInk places instances sequentially, fills the page, and creates additional pages automatically when needed.

2.6 A first taste of the DSL

Once basics work, you can start controlling page and layout with a short expression:

```
{A4 b=[-5]}.L{p=4x3 g=2}
```

At a glance:

- `{A4 b=[-5]}`: A4 page with 5 mm inner margin at every side.
- `.L{p=4x3 g=2}`: layout of cards in a pattern of 4x3 grid with 2 mm gap.

This short notation is part of the PnPInk DSL (Domain Language). It lets you control placement, scaling, rotations, grids, gaps, bleeds, marks, and more.

PnPInk is designed to be simple by default, and powerful when you need it.

2.7 What PnPInk can do (quick tour)

Even if you do not understand every syntax detail yet, this gives you a practical map of what is possible.

2.7.1 From simple repetition...

Build many components from one design and one dataset:

```
{A4}.L{3x4}
```

One template can be placed 12 times on an A4 page. See Layout and Page.

2.7.2 ...to data-driven variation

Each dataset row can produce a different result. Texts, images, icons, and properties can vary per instance. See Dataset Reference.

2.7.3 Precise layout control

Control spacing and sizing explicitly:

```
L{3x4 gaps=4 shape=poker}
```

See Layout.

2.7.4 Fronts and backs, automatically

Generate aligned duplex backs with `@back`:

```
{card_back @back}
```

See DSL Advanced.

2.7.5 Page-level elements

Place objects once per page, not once per card:

```
{page_title @page}
```

Useful for titles, page numbers, frames, or static page backgrounds. See Page.

2.7.6 Fit and Anchor (single concept)

Position objects relative to target rectangles without manual coordinates:

```
icon.F{i a=9}
```

The element is fitted and anchored by intent, not by absolute measurements. See Fit and Anchor.

2.7.7 Adaptive layouts

Let layout adapt to available space:

```
L{1x? gaps=?}
```

Items stack and spacing is computed automatically. See Layout.

2.7.8 Explicit ID arrays and slots

Apply layout directly to selected IDs:

```
[id1 id2 - id3].L{1x?}
```

`-` reserves an empty slot without rendering. See Core Syntax.

2.7.9 Slot-level alignment

Align content inside each layout slot:

```
L{1x? a=6}
```

See Layout and Fit and Anchor.

2.7.10 Production features

Generate cut marks aligned with final geometry:

```
.M{len=[3 2] d=2}
```

Marks follow real layout, spacing, bleeds, and rotations. See Marks.

Use external sources (images, PDFs, spritesheets, icon libraries), and reuse generated assets in pipelines. See Source.

2.8 A key practical advantage

With PnPInk you do not need to work with absolute coordinates for most production tasks.

You can express intent, for example: bring an image from a source, rotate it, fit it to a top-right anchor, scale it, and crop overflow.

If the source image changes, the same rule still works. If you switch card size (for example from poker to tarot), layout and fitting scale with the template logic, without manually re-measuring everything.

This is one of the main differences between PnPInk and many manual or coordinate-heavy workflows.

2.9 Recommended reading path

1. Basic Workflow
2. Dataset Format and Sources
3. Dataset Reference
4. DSL Nomenclature
5. DSL Modules
6. Fit and Anchor

Dataset

3.1 Dataset Format and Sources

PnPInk reads tabular data from CSV files or Google Sheets and converts it into one or more dataset sections.

3.1.1 Supported Inputs

- Local CSV file.
- Google Sheet (`sheet_id` in DeckMaker).

If `sheet_id` is empty, DeckMaker loads CSV from the same folder as the SVG:

```
<svg_name>.csv
```

3.1.2 Google Sheets Setup

In `Extensions > PnPInk > DeckMaker`, fill:

- `GSheet ID`
- `Sheet!range/gid` (optional)

How to get `sheet_id`:

- From a Google Sheets URL like: `https://docs.google.com/spreadsheets/d/<SHEET_ID>/edit#gid=0`
- Use the part between `/d/` and `/edit`.

3.1.3 Google Sheets Access Modes

When `GSheet ID` is set, PnPInk supports two access modes:

1. Public access (no OAuth). Use numeric `gid` in `Sheet!range/gid` (example: `381688145`). If this field is empty, PnPInk uses `gid=0`.
2. Private access (OAuth, authenticated). This is the more secure mode. If `Sheet!range/gid` is empty, PnPInk tries sheet name = SVG filename, then falls back to the first sheet.

You can also set an explicit selector:

- `SheetName!A1:Z99`

3.1.4 Dataset Section Structure

Each dataset section has:

- column A: section marker and row-level control cell,
- columns B+: headers and data fields.

Two modes are supported:

- Marker mode: `{{...}}` in column A (recommended, supports multiple dataset sections).
- Shorthand mode: single dataset; first row uses A1 as main bbox id.

See full syntax in Dataset Reference.

Minimal Dataset Example

Column A	title	cost	art
`card_bbox`			
	Fireball	3	images/fireball.png
	Shield	2	images/shield.png

```
card_bbox,title,cost,art
,Fireball,3,images/fireball.png
,Shield,2,images/shield.png
```

3.1.5 Why Column A Is Critical

The first bbox id (`t=...` or equivalent) in column A defines the main template for that section. It drives:

- layout slots and pagination,
- marks generation,
- front/back slot pairing,
- page-membership selectors for `@page`.

3.1.6 Headers, Comments, and Directives

Headers and comments are part of the dataset grammar, not free text. Before authoring complex sheets, review:

- Dataset Reference for exact rules,
- Snippets for `# :Name(...) = ...` directives,
- DSL Nomenclature for shared token rules.

3.1.7 Multi-Dataset and Multi-Template

- Multiple datasets in one sheet are supported via repeated marker rows (`{{...}}`).
- Multiple template columns are supported via header declarations (`{template_id ...}`).
- Multiple main templates in one marker list are not supported; use one main template per dataset section.

3.2 Dataset Reference

This chapter is the exact behavioral reference for dataset parsing and execution. Use Dataset Format and Sources first if you are new to the model.

3.2.1 IDs and Naming

This section explains how table headers bind to SVG content.

Dataset headers are matched to SVG IDs.

For the Inkscape ID workflow (where to inspect and edit IDs), see Introduction -> How IDs connect data to graphics.

IDs must be unique and follow XML rules (letters first, no spaces).

3.2.2 Dataset Structure

Dataset structure defines where parsing starts and how sections are separated.

PnPInk supports two forms:

- **Marker mode** (recommended, supports multiple datasets).
- **Shorthand mode** (single dataset only).

For source selection (CSV vs Google Sheets and tab lookup), see Dataset Format and Sources.

Marker Mode (column A)

Marker mode is the robust format for production datasets and multi-section files.

A dataset marker exists **only in column A** and uses `{{...}}`. The marker row is also the header row. Headers start in column B.

Examples (equivalent):

```
{{t=card_bbox}}
{{template_bbox=card_bbox}}
{{card_bbox}}
```

Notes:

- Only one main template bbox is supported per dataset marker.
- Extra DSL tails are allowed after the marker (see Leading Cell below).

Main Template BBox and Z-Order

This is a core concept: one main bbox drives slot logic for the section.

The marker `t=...` defines the **main template bbox** for that dataset section. This main template controls:

- slot planning (`Layout{}` / `Page{}`),
- per-slot marks (`Marks{}`),
- front/back slot pairing (`@back`),
- page membership for page-anchored templates (`@page` selectors).

Placement order (Z-order) follows dataset row order. Later rows are rendered above earlier rows.

Shorthand Mode (single dataset)

Shorthand mode is a convenience form for small, single-section sheets.

If there is no marker row, the first non-empty, non-comment row is treated as the header row.

In shorthand mode, column A contains the template bbox id:

```
card_bbox, title, cost, art
```

This is equivalent to:

```
{{t=card_bbox}}, title, cost, art
```

3.2.3 Header Types

Header type determines whether a column edits fields or instantiates templates.

Headers in column B+ can be:

1. **Normal data fields**: match SVG IDs and replace text or sources.
2. **Template columns**: declare extra templates with `{...}`.

Normal Data Field Syntax

Use this syntax for the most common per-row updates (text, source, and defaults).

Headers can include modifiers:

```
title
art+
price[xml]
card_bg[fill]
line[stroke]
bg=default_bg
art=art_placeholder~i5
```

Rules:

- `id+` keeps the original anchor rect visible (otherwise anchors are hidden).
- `id[prop]` sets a property. Default is `text`.
- `id=...` declares a **default value** or default Fit ops for that column.
- `id-*` in headers expands to all matching IDs by prefix.
- `id1 id2 id3` in a single header applies the same cell value to all listed IDs.

Examples:

```
id=default_id
id=~i5
id=default_id~i5
```

Style Property Columns

Use `id[property]` to change SVG style properties from the dataset.

```
card_bg[fill]
card_bg[stroke]
card_bg[stroke-width]
title[fill]
title[font-size]
line[stroke]
```

Examples:

```
card_bg[fill],title[fill],line[stroke],line[stroke-width]
#f4ead2,#12110f,b8a300ff,2
```

Style columns update the target element's `style` attribute. When a style property is changed, PnPInk keeps that element visible automatically; you do not need to add `+`.

Use `fill` for closed shapes and text color. Use `stroke` for lines, open paths, outlines, and most visible path strokes.

Color values can be written as normal SVG values or compact hex:

```
#ff0000
ff0000
#ff000080
ff000080
red
url(#myGradient)
```

For `fill` and `stroke`, 8-digit hex colors are split into color plus opacity for SVG compatibility. For example, `ff000080` becomes red with partial `fill-opacity` or `stroke-opacity`.

Property-only shorthand inherits the previous target id:

```
card_bg[fill],[stroke],[stroke-width]
#f4ead2,#222222,0.4
```

This is equivalent to:

```
card_bg[fill],card_bg[stroke],card_bg[stroke-width]
```

Special properties:

- `id[text]` or just `id` replaces text.
- `id[xml]` replaces rich XML content inside text-like objects.

Header Fan-Out and Wildcards

Use this when one dataset column must feed several placeholders.

Examples:

```
ph-1 ph-2 ph-3
id1
```

`id1` is applied to all three placeholders.

Wildcard headers are also supported:

```
main_icon-*
Ic(heart-suit)
```

This applies the value to every placeholder whose ID starts with `main_icon-`. All normal header modifiers (`+`, `[xml]`, `=...`) remain valid.

Template Column Syntax

Use template columns when you need extra template instances, back passes, or page-level elements.

Template columns declare template bbox IDs and modifiers:

```
{card_back @back}
{page_title @page}
{back_bg @page @back}
```

Supported modifiers:

- `@page` page-anchored templates
- `@back` back-pass templates

Template columns are rendered as additional instances; they do not replace the main template column logic.

3.2.4 Comments and Directives (#)

Comments are processed **before any other operation**. Rules are global (same behavior everywhere; no inside/outside distinction).

- `#` at line start (first non-space char in column A): comment/directive row.
- `##` at line start: full comment row (not a directive).
- `####` at line start: EOF marker, stops parsing the rest of the file/sheet.
- `##` inside a cell: comments out the rest of that cell.
- `###` inside a row (not line start): comments out the rest of that cell and all cells to the right (rest of line).
- Single `#` inside a cell is normal text (not a comment).

Header disabling still works:

- `##header` disables that column.
- `###header` disables that column and all columns to the right.

3.2.5 Leading Cell (column A in data rows)

Leading-cell directives are row-level controls, not regular data fields.

Column A in data rows can carry row-level DSL:

- `{A4 ...}` page block
- `L{...}` layout tail
- `M{...}` marks tail
- trailing copies number
- optional hole patterns in `[...]`
- optional iterator selection in `[...]` when using numeric ranges like `1..5 7..100`

Examples:

```
{A4 b=[-5]} L{p=3x3 g=2} M{mk_cut} 2
[3 - 2-]           -> 3 copies, then 1 hole, then 2 holes
[1..5 7..100]      -> keep iterator items 1..5 and 7..100 (skip 6)
[1..4 3- 7..9]     -> keep 1..4, then 3 holes, then keep 7..9
```

Hole syntax in the final `[...]`:

- `-` = 1 hole after the current copy count
- `N-` = `N` holes after the current copy count
- plain numbers add copies before later holes are placed

Examples:

```
[3 - 2-]    -> 3 copies, then 1 hole, then 2 more holes
[2 3- 5]    -> 2 copies, then 3 holes, then 5 more copies
```

When the final `[...]` contains numeric ranges (`..`), it filters row iterators (`*[...]`). Hole markers (`-`, `N-`) can still be mixed in the same block and are applied after the accumulated selected run.

This cell is **not** a normal dataset field; it controls row-level layout/flow. Its directives apply before regular field replacements in that row.

For the full sequencing rules of column A with iterators, copies, holes, reordering and `?`, see Iterators, section **Row Sequencing from Column A**.

If a row uses only column-A controls and all payload cells (columns B+) are empty, PnPInk applies the controls but does **not** generate a card/instance for that row.

```
card_bbox,title,cost
{A4},,
,Fireball,3
```

3.2.6 @back -- Back-Side Templates (Back Pass)

Use `@back` for duplex output where back faces must align with front slot geometry.

A template column marked with `@back` is rendered **only on back pages**.

```
{card_back @back}
```

Back templates:

- reuse the same slot sequence from the front layout,
- mirror slot placement within the page for duplex alignment,
- preserve row order as Z-order.
- back pages are inserted right after each front page (interleaved).

If a dataset cell for an `@back` column is empty or contains `-` or `0`, that back instance is skipped.

3.2.7 @page -- Page-Anchored Templates (One Per Page)

Use `@page` for elements that belong to the page frame, not to per-card slots.

A template column marked with `@page` is anchored to the **page frame**, not to the slot grid.

```
{page_title @page}
```

Page-anchored templates:

- are positioned relative to the page frame after `Page{}` margins,
- are rendered once per page,
- use Fit/Anchor for placement.

Each dataset row provides a **slot selector** in the cell value (e.g. `~8[-5]`). The slot determines which page the template belongs to; the rest is Fit/Anchor ops.

Selectors can be a single slot index, range, or list:

```
~1
~[1 3 5]
~[2..4]
```

If multiple rows target the same page, only the first is rendered (a warning is logged).

3.2.8 Combining @page and @back

Combining both modifiers is common for back-page backgrounds and page-level back labels.

```
{back_bg @page @back}
```

This places a page-anchored element on back pages, aligned to the front page sequence.

3.2.9 Advanced: Split Boards

This mode is activated automatically when template dimensions exceed target page inner size.

If the template is larger than the target page, the engine switches to split-board mode:

- the template is cut into tiles,
- each tile is placed on a page,
- layout and marks are applied to the tile bounds.

4 ZVG and PNP Packages

PnPInk supports two ZIP-based package formats for portable projects:

- `.zvg`: package the current SVG project and local assets for reproducible sharing.
- `.pnp`: package a lightweight project intended to regenerate content from dataset/web sources on open.

4.1 Where They Appear in Inkscape

- `File > Open` can open `.zvg` and `.pnp`.
- `File > Save As` / `Save a Copy` can export `.zvg` and `.pnp`.

4.2 Import Extraction Folder

When opening a `.zvg` or `.pnp`, PnPInk extracts package contents into a dedicated folder:

- folder path: same directory as the package file,
- folder name: package filename without extension.

Example:

- package: `MyDeck.pnp`
- extracted to: `MyDeck/` (contains SVG, manifest, CSV, assets, etc.)

This keeps assets from different packages isolated and avoids cross-package mixing.

4.3 Package Contents

Typical package structure:

- One main SVG in ZIP root.
- Optional CSV dataset.
- Optional `manifest.json`.
- Local assets (typically under `assets/`).

Defaults work without manifest:

- If there is one SVG in root, it is used as main document.
- CSV is expected with the same base name as the SVG.
- Assets are expected under `assets/`.

If multiple SVG files are included, set `manifest.json` with the `svg` field to disambiguate.

4.4 ZVG vs PNP Behavior

- `.zvg` favors reproducibility:
- includes referenced local assets,
- includes local cached downloadable assets when referenced.
- `.pnp` favors minimal payload:
- includes required local project assets,

- skips cache-like downloaded files when possible,
- if local CSV is missing and `gsheet_id` is available (options or manifest), export generates a CSV snapshot from Google Sheets and includes it,
- may run DeckMaker on import when requested by manifest (`run_deckmaker_on_import`).

4.5 Compression Policy

- Images and PDFs are stored without deflate recompression:
- `.png`, `.jpg`, `.jpeg`, `.webp`, `.pdf`
- Other files are compressed with deflate (level 9).

4.6 Recommended Use

- Use `.zvg` to archive/share an exact visual state.
- Use `.pnp` to share a compact, regeneration-oriented project.

DSL

5.1 DSL Nomenclature

This chapter defines the shared language used across the DSL (Domain Language). Use it as the common reference before reading module-specific pages.

5.1.1 What belongs here

PnPInk has two complementary parts:

- dataset structure (rows, headers, section markers, control columns),
- DSL logic (declarative syntax that controls how data is interpreted and rendered).

This chapter is about the second part: the DSL itself.

In practice, this includes:

- module syntax (`Fit`, `Layout`, `Page`, `Source`, `Marks`, `Gradients`),
- compact notation (`~...` and operator shorthands),
- list/range/repetition grammar,
- global directives in comment lines (`# ...`) such as snippets, spritesheet aliases, and other declarations.

For dataset structure and section markers, see Dataset Reference.

5.1.2 IDs and Targets

An SVG `id` is the primary selector for DSL operations.

For the Inkscape ID workflow (where to inspect/edit IDs and structure), use: Introduction -> How IDs connect data to graphics.

Target forms:

- one SVG object id: `title`,
- grouped target list: `[id1 id2 id3]`,
- source target: `@{assets/icon.svg}`.

These selectors let one logical target expose multiple concrete items without creating many separate headers.

5.1.3 Module Form

Modules define behavior declaratively. Use long form for clarity and short form for compact production datasets.

Long form:

```
target.Module{ key=value key2=value2 }
```

Here, `key` and `key2` are module **parameters**. Each module defines its own valid parameter set, defaults, and aliases. In this documentation, we always call them **parameters** (not properties).

Short form:

```
target.M{ ... }    # module short name
target~...         # Fit-Anchor shorthand
```

Common modules:

- `Fit{}` / `~...`
- `Layout{}` / `L{}`
- `Page{}`
- `Source{}` / `@{}`
- `Marks{}` / `M{}`

5.1.4 Global Tokens

- `[]` list or list-like values.
- `{}` module blocks (`Page` , `Layout` , `Marks` , etc.).
- `^` rotation (and page landscape in page tokens).
- `|` and `||` mirror (horizontal, vertical).
- `!` clip (see Fit-Anchor clip stage details).
- `:` substitution (inline `:icon:` and `:snippet(...)`).
- `*` has three different uses depending on context:
- wildcard in identifiers/paths (`id*` , `file*.png`),
- iterator prefix in dataset values (`*[ ... ]` , `**[ ... ]` , ...),
- repetition inside list/array bodies (`3*id` , `3*(id1 id2)`).
- `()` function arguments (`:snip(a b)`) and group blocks inside lists/iterators.

5.1.5 Lists, Ranges, and Repetition

Lists are space-separated:

```
[ID1 ID2 ID3]
```

Supported patterns in list/array bodies:

```
[ID1 - - - ID4]    -> same as [ID1 3- ID4], '-' means empty slot
3*ID               -> [ID ID ID]
5*:Ic(potato)      -> [:Ic(potato) :Ic(potato) :Ic(potato) :Ic(potato) :Ic(potato)]
[id1 3*(id2 5- id3)]
```

Repetition:

- `n*Id` means repeat `Id` `n` times.
- Example: `[3*Id]` is equivalent to `[Id Id Id]` .
- `Id` can be a `( )` grouped block: `[2*(idA idB)]` is equivalent to `[idA idB idA idB]` .

Ranges and selectors:

```
target[2]      -> item 2
target[2..4]   -> items 2, 3, 4
target[A..ZZ]  -> A B C ... Z AA AB .. ZZ
target[*]      -> all items
target[2][3]   -> chained selector for nested/indexed structures
```

Note that `target[*]` (selector) and `*[ ... ]` (iterator) are not the same operation. - `target[*]` selects all items inside one target expression. - `*[ ... ]` expands rows/instances during iterator processing (asign items to diferent cards)

5.1.6 Units and Expressions

In paramters, most numeric tokens accept:

- plain numbers (current default unit),
- `mm`, `px`, `%`,
- mixed expressions such as `-25%+2`.

Percentages are evaluated against a base size. For example border=[80% 110%+3] define a new rect size with 80% of actual width, and 110% of height + 3 mm.

5.1.7 Comment Directives and Declarations

Comment rows (`# ...`) can declare DSL directives (aliases, variables, spritesheets...) outside normal data rows. Typical examples:

```
# :snippet(x) = ...
# @sp1 = @{sheet.png}.Layout{8x6}
# gradientGold = G{b88326ff^25 [-25 12 5] [55 35 20 10]}
# _DM_reset_to=3
```

These declarations are read by their corresponding modules/subsystems and then reused in dataset rows.

5.1.8 Abbreviations, Defaults, and State Reuse

Keyword abbreviations are supported with the firs character.

```
Layout{} -> L{}
inside -> i
Page{pagesize=A4 border=[-2]} -> P{A4 b=[-2]}
```

Most modules define default parameters, so compact forms can omit explicit keys. I.e. you dont need to specify p=A4 because "pagesize" is the default parameter por Page{} module.

When a required value is omitted, the last active state may be reused (module-dependent behavior).

For exact page-state rules, see Page. For Fit-Anchor shorthand order and precedence, see Fit-Anchor.

5.2 DSL Architecture

High-level flow of how PnPInk applies the DSL.

5.2.1 Data to SVG Pipeline

This sequence explains where each DSL family acts in the generation process.

PnPInk processes in this order:

1. Read dataset (CSV or Google Sheets).
2. Parse dataset marker and header row.
3. Clone the template for each row (main template column).
4. Resolve sources (images/icons/URLs) into SVG symbols.
5. Apply Fit/Anchor to place each element.
6. Apply Layout to place instances into slots and pages.
7. Render Marks (cut marks) per placed slot.

5.2.2 Front, Back, and Page Passes

These passes are execution phases, not just syntax flags. They define when and where templates are rendered.

Header modifiers control when a template is rendered:

- `@page`: page-anchored, once per page, positioned against the page frame.
- `@back`: rendered in the back pass, aligned to front slots.
- `@page @back`: page-anchored but rendered on back pages.

These modifiers belong to template headers, not to data cells.

5.2.3 Where Layout and Page Apply

This distinction prevents common mistakes when debugging placement.

- `Page{}` defines page size, margins, and cursor position.
- `Layout{}` defines the slot grid and how instances are placed.
- `Fit{}` defines how elements sit inside their anchor rects.

Page state is global. Layout state is per dataset. Fit is per element.

5.2.4 Sources

Sources are resolved once and then treated as normal placement targets.

`Source{}` / `@{}` creates a reusable SVG symbol and then places it with Fit, so sources behave like normal SVG targets.

5.2.5 Inkscape Integration

These panels are the operational bridge between visual authoring and dataset-driven generation.

PnPInk relies on Inkscape objects and IDs:

- IDs and structure workflow: Introduction -> How IDs connect data to graphics.
- Symbols: `Object > Symbols` (Shift+Ctrl+Y).
- Layers: `Layer > Layers` (Shift+Ctrl+L).

5.3 Iterators and Copies

This chapter explains the difference between array placement and row iteration, and how both interact with card quantity in column A.

5.3.1 Why this matters

At first sight, these two expressions look similar:

```
[id1 id2 ... idn]
*[id1 id2 ... idn]
```

They are not equivalent. The first builds one composite placement inside one generated card. The second expands the dataset row into multiple generated cards, one per iterator value.

5.3.2 Array Without Iterator (`[ ... ]`)

When a non-text field uses:

```
[id1 id2 ... idn]
```

PnPInk treats it as an array/group target for that field in the current card instance. All listed items are processed in the same row instance.

This is useful when you want several IDs resolved together in one card, and optionally apply local array layout/fit behavior.

Wildcard IDs are supported inside arrays:

```
[main_icon-*]
```

This expands to all matching IDs and keeps array/list semantics (single card instance).

Array repetition shorthand is also supported:

```
[3*:Ic(potato)]~i2
```

Equivalent to writing the same token three times in the list.

Grouped repetition is also supported with parentheses:

```
[id1 - 3*(id2 3- 3*id3) 4- id4]
```

Parentheses are grouping operators inside list/array bodies, so you can repeat a block as one unit.

5.3.3 Iterator (`*...`)

When a cell starts with `*`, it becomes an iterator expression.

```
*[id1 id2 ... idn]
```

Result:

- the row is expanded into multiple internal instances,
- each generated card gets one iterator value at that cell,
- by default (without explicit copies), generated cards count equals iterator length.

Conceptually: one different card per value.

Parentheses inside iterator lists

Inside `*[ ... ]`, parentheses define one grouped iterator item (multivalue in the same generated card instance):

```
*[id1 id2 id3]
```

This iterates 3 scalar items (`id1`, then `id2`, then `id3`).

```
*[id1 (id2 id3)]
```

This iterates 2 items:

- first iteration: `id1`
- second iteration: `id2 id3` (multivalue group in the same card)

So `*[id1 id2 id3]` and `*[id1 (id2 id3)]` are intentionally different.

Group repetition also works:

```
*[id1 2*(id2 id3)]
```

Equivalent iterator sequence: `id1`, `(id2 id3)`, `(id2 id3)`.

Wildcard IDs are also supported in scalar/multivalue cells:

```
main_icon-*
```

Outside `[ ... ]`, wildcard expansion behaves as multivalue (same card instance, token sequence order).

5.3.4 Multi-Level Iteration (`*`, `**`, `***`)

PnPInk supports multiple iterator levels using leading stars.

```
*[A B C]
**[1 2]
```

This produces a nested expansion (cartesian-style): every level-1 value is combined with every level-2 value. You can add more levels (`***`, etc.) when needed.

5.3.5 Supported Iterator Expressions

Current parser supports iterator payloads such as:

- bracket list/range: `*[1 2 3]`, `*[1..10]`, `*[A..F]`,
- virtual sources with selectors,
- filesystem glob through `*@{...}`,
- spritesheet wildcard alias: `*@alias[*]`.

If an iterator resolves to zero values, that row yields zero generated instances.

5.3.6 Row Sequencing from Column A

Column A can control the final sequence of generated cards for a row. This is broader than just "number of cards": it can skip execution, set a fixed count, filter iterator positions, reorder them, and insert empty slots.

Think of the pipeline like this:

1. Expand row iterators (`*[...]`, `**[...]`, source iterators, etc.).
2. Apply any selector written in the final column-A `[...]`.
3. Apply explicit copies if present.
4. Insert holes after the selected/generated sequence positions.

Default behavior

If column A does not specify copies:

- without iterators: the row generates `1` card
- with iterators: the row generates the full iterator length

Examples:

```
Column A: <empty>
art = icon_fire
```

Interpretation: one card.

```
Column A: <empty>
art = *[A B C]
```

Interpretation: three cards -> `A`, `B`, `C`.

`0` means "do not generate this row"

If column A resolves to `0` copies, the row is skipped completely.

```
Column A: 0
art = *[A B C]
```

Interpretation: no cards are generated from that row.

Explicit copies

Column A can declare an explicit quantity.

With iterators:

1. If copies > iterator length, values wrap around.
2. If copies < iterator length, the sequence is truncated.

Examples:

```
Column A: 5
art = *[A B C]
```

Interpretation: `A, B, C, A, B`.

```
Column A: 2
art = *[A B C]
```

Interpretation: `A, B`.

Selector syntax in the final `[...]`

The final bracket block in column A can select iterator positions explicitly.

Examples:

```
[1..5 7..100]
```

Interpretation: - keep `1,2,3,4,5` - skip `6` - then keep `7..100`

```
[3..1 4..5]
```

Interpretation: - reorder the beginning as `3,2,1` - then continue with `4,5`

This selector is applied to the expanded iterator sequence before copy wrapping/truncation.

Unknown end: `?`

Use `?` to mean "the final iterator position calculated by PnPInk".

Examples:

```
[13..?]
```

Interpretation: keep from `13` to the last iterator item.

```
[?..12]
```

Interpretation: keep from the last iterator item down to `12`, in reverse order.

Empty slots (holes)

Hole syntax uses:

- `-` = 1 empty slot
- `N-` = `N` empty slots

Holes are inserted **after the accumulated generated run at that point**.

Examples:

```
[3 - 2-]
```

Interpretation: - generate 3 cards - then insert 1 empty slot - then insert 2 more empty slots

```
[2 3- 5]
```

Interpretation: - generate 2 cards - insert 3 empty slots - then generate 5 more cards

Mixing selection, reordering and holes

Selection and holes can be mixed in the same final `[...]`.

Examples:

```
[1..4 3- 7..9]
```

Interpretation: - keep iterator items `1,2,3,4` - insert 3 empty slots - keep iterator items `7,8,9`

```
[3..1 2- 4..5 7..9 ?..12]
```

Interpretation: - reorder the first block as `3,2,1` - insert 2 empty slots - keep `4,5` - keep `7,8,9` - then append from the last item down to `12`

Practical summary

Column A can therefore be used to:

- leave default generation untouched
- skip a row with `0`
- force a fixed number of copies
- skip parts of an iterator sequence
- reorder iterator positions
- refer to the unknown end with `?`
- insert empty slots between selected/generated runs

5.3.7 Examples

One card with grouped IDs

```
art = [icon_fire icon_air icon_earth]
```

Interpretation: one row instance, one generated card, grouped placement in that card.

One card per ID

```
art = *[icon_fire icon_air icon_earth]
```

Interpretation: three generated cards from that row (unless column A copies overrides).

Iterator + explicit copies

```
Column A: 5
art = *[A B C]
```

Interpretation: 5 cards -> `A, B, C, A, B` (wrap).

```
Column A: 2
art = *[A B C]
```

Interpretation: 2 cards -> `A, B` (truncate).

5.4 Fit-Anchor

Fit-Anchor is one of the most important modules in PnPInk. It is the placement engine that lets you position objects relative to placeholders without manual coordinates and without hardcoding object/template sizes.

In practice, Fit-Anchor lets you place one or many objects on a placeholder and then size, align, shift and clip them relative to that placeholder.

Fit-Anchor is about the relationship between an object and its placeholder. For visual changes applied to the object itself after placement, use Transform.

5.4.1 Dataset Mental Model

Fit-Anchor is easier to understand if you keep this model in mind:

- the dataset header points to a placeholder object in the SVG,
- each row cell provides one or many source objects,
- Fit-Anchor defines how those source objects are positioned relative to that placeholder for each generated card/instance.

5.4.2 Fit vs Transform

Use:

- `Fit` / `Fit-Anchor` for placement relative to the placeholder
- `Transform` for changes applied to the placed object itself

Typical Fit-Anchor concerns are:

- size inside the placeholder
- alignment within the placeholder
- border or fit area
- clipping to the placeholder

Typical Transform concerns are:

- opacity
- soft edges
- later, other visual effects applied directly to the object

5.4.3 What Is an "Object" Here?

In this documentation, we call IDs "objects" (not "elements") from the DSL/user point of view.

An object can be:

- an existing SVG object by ID (rect, path, group, text, image, etc.),
- a dynamic object created by `Source` (local file, spritesheet frame, internet URL, Wikimedia/Pixabay, icon library),
- a multivalue object list in one cell: `id1~7 id2~9 id3~3`,
- an explicit object array: `[id1 id2 id3]` (optionally with local array layout).

Representative examples:

```
main_art-8
heart_icon
@{assets/picture.png}~i5
@sp1[14]~m5
@{wkmc://File:Example.svg/svg#nodeX}~i5
```

```
id1~7 id2~9 id3~3
[id_a id_b id_c].L{3x1 g=2}~i5
```

5.4.4 Syntax and Style

Fit-Anchor follows the same DSL conventions as other modules:

- `Module{ key=value ... }` long form,
- short aliases (`a` , `b` , `s` , etc.),
- default parameters can be omitted,
- compact shorthand with `~` is available and heavily used in real datasets.

Long form:

```
object_id.Fit{ border fitmode anchor shift clip rotate mirror }
```

Compact form:

```
object_id~...
```

Compact Example (Explained)

Example:

```
id1~[10%]i7^^[-50% 0]!
```

Read it left to right:

- `[10%]` -> border (resize context around placeholder),
- `i` -> fit mode `inside` ,
- `7` -> anchor top-left,
- `^^` -> rotate 180 deg,
- `[-50% 0]` -> shift left by half placeholder width,
- `!` -> clip to the placeholder shape.

This compact style is very expressive once you know the token order.

5.4.5 Fit-Anchor Parameters

The following subsections describe each Fit-Anchor parameter and what it changes in final placement. Order matters conceptually.

Border (`b=`)

Border changes the usable fit area around the placeholder.

- positive values expand,
- negative values shrink.

Important behavior:

- `border` modifies placeholder size **only for Fit Mode scaling**.
- `anchor` , `shift` , and `clip` still use the **original placeholder** as geometric reference.

In other words:

1. border adjusts the fit area,
2. fit mode computes object scale in that adjusted area,
3. anchor/shift/clip are evaluated against the original placeholder.

This is why defining `border` first is conceptually important: it controls object size before final positioning.

Examples:

```
border=[-2]
border=[2 3]
border=[2 3 4 5]
object_id~[-2]i
```

Percent and absolute-size forms are supported:

```
border=[50%]
border=[125%]
border=[40x60]
border=[50%x20]
border=[23x?]
border=[?x12]
```

Negative `WxH` components can flip orientation:

```
border=[100%x-100%]
border=[-100%x100%]
```

Fit Mode

Fit mode defines scaling behavior relative to the (possibly border-adjusted) fit area. If omitted, `inside` is the practical default.

Examples:

```
object_id.Fit{fitmode=inside}
object_id~i
```

Fit Mode Reference

Code	Name	Description
i	inside / contain	Scales proportionally to fit entirely within the rect (default).
o	original / none	Keeps original size.
w	width-fit	Scales proportionally to match rect width.
h	height-fit	Scales proportionally to match rect height.
m	max / cover	Scales proportionally until it covers the rect, possibly overflowing.
x	x-stretch	Stretches width to match rect (non-proportional).
y	y-stretch	Stretches height to match rect (non-proportional).
a	all-stretch	Scales independently in X/Y to fill rect exactly.
t	tile	Tiles the object as a pattern within the rect.
b	best-fit	Smart mode that mixes m, a, and clipping for balance.
?	auto-fit	Alias of `b` (best-fit).

Anchor

Anchor selects the reference point used to align an object to the placeholder. Think of it as: "which point of the object goes to which point of the placeholder."

Anchor uses the **original placeholder** (not the border-adjusted fit area). This makes anchor behavior stable and predictable while `border` only affects scale.

`anchor = 1..9` follows numeric keypad:

7 8 9

4 5 6

1 2 3

Examples:

```
object_id.Fit{anchor=7}
object_id~7
```

If no explicit anchor is provided, center (`5`) is used as default.

Shift (`shift=` / `s=`)

Shift offsets final position after anchor/fit.

```
object_id.Fit{anchor=7 shift=[-100% -100%]}
object_id~i8[0 %]
object_id~i8[-50% 0]
```

Notes:

- `%` is relative to placeholder size,
- `%` means `100%`,
- `-%` means `-100%`,
- mixed expressions are valid (`shift=[25%+2 0]`).
- shift uses the original placeholder frame as reference.

Rotate (`r=` / `^`)

Rotates the target object.

```
object_id.Fit{rotate=-42.4}
object_id~^^^
object_id~^-45i7
```

Mirror (`m=` / `|`)

Mirrors the target.

- `|` horizontal,
- `||` vertical.

```
object_id.Fit{mirror=v}
object_id~||
```

Clip (`c` / `!`)

Clips outside the original placeholder shape.

```
object_id~i8![0 %]   # pre-clip then shift
object_id~i8[0 %]!   # shift then post-clip
```

Position of `!` matters.

5.4.6 Compact Token Order (Reference)

When using shorthand `~`, tokens are parsed in this order:

1. Optional border list (`[t r b l]`, `[x]`, `[x y]`)
2. Fit mode + anchor (`i7`, `m5`, `a9`, etc.)
3. Optional shift list (`[dx dy]`)
4. Optional clip (`!`)
5. Optional rotation (`^deg`, `^^`, `^^^`)
6. Optional mirror (`|`, `||`)

5.4.7 Priority and Overrides

When multiple Fit-Anchor layers apply, use this precedence:

1. Header defaults/global ops
2. Iterator/global ops
3. Item-local ops

Later layers override earlier ones for conflicting properties.

Example:

```
Header:
main_art-8=~[10x10]

Cell:
*[:fig(g918)~^^ :fig(g1720) :fig(g1264) :fig(g13590)]~[0%]5
```

Effective merge:

- start from header border,
- apply iterator-level defaults,
- apply item-level overrides last.

5.4.8 Practical Examples

This section intentionally groups compact real-life patterns.

5.4.9 Compact Notation (Clarifications)

This section consolidates the compact syntax details that usually cause confusion.

With `~` and without `~`

For simple rotate/mirror/clip operations, `~` can be omitted:

- `id~^15` and `id^15` are equivalent.
- `id~!` and `id!` are equivalent.
- `id~|` and `id|` are equivalent.
- `id~||` and `id||` are equivalent.

Use the `~` form when you want the full compact chain in one expression (`border + fit/anchor + shift + clip + rotate + mirror`).

Combined compact operations

You can combine these operators in one token, for example:

```
id~i5^^!
id^90|
id~m7[0 5%]!
```

Parsing and precedence reminder

- The compact parser still applies the same Fit-Anchor merge precedence: header defaults -> iterator/global ops -> item-local ops.
- For conflicting properties, the last layer overrides previous ones.

Example 1: Simple inside-center

```
art_id~i5
```

Places object inside placeholder, centered.

Example 2: Cover and clip

```
art_id~m5!
```

Covers placeholder area, then clips overflow to placeholder shape.

Example 3: Corner icon with offset

```
icon_id~i7[2 -2]
```

Inside fit, top-left anchor, then fine offset.

Example 4: Multiple objects in one cell

```
id1~7 id2~9 id3~3
```

Places three objects in one placeholder flow, each with its own anchor.

Example 5: Array group with local layout

```
[id1 id2 id3].L{3x1 g=2}~i5
```

Builds an array object, lays it out locally, then applies Fit-Anchor as one grouped target.

5.4.10 Related Pages

- Transform
- Source

5.5 Transform

`Transform` applies visual changes to the object itself after it has already been placed.

Use it when you want to alter the final appearance of an object without changing the placeholder logic of `Fit` or `Fit-Anchor`.

`Transform` is not about the placeholder. It is about the object itself once placement is already solved.

5.5.1 Transform vs Fit

Use:

- `Fit` / `Fit-Anchor` for anything relative to the placeholder
- `Transform` for anything applied to the object itself

Examples of `Fit` concerns:

- fit mode
- anchor
- border
- clipping to the placeholder
- placement inside the placeholder

Examples of `Transform` concerns:

- opacity
- soft edges
- later, other visual effects such as blur, color adjustments or shadows

5.5.2 Syntax

```
object_id.T{opacity=50% soft=12%}
object_id.T{o=50% s=12%}
object_id.T{f=myFilter}
object_id.T{f=image1-9-1 s=8%}
@{source}.T{o=70%}
```

`Transform` can be written as:

- `Transform{...}`
- `T{...}`

5.5.3 Parameters

Current parameters are:

- `opacity` or `o`
- `soft` or `s`
- `filter` or `f`

5.5.4 Opacity

```
.T{o=50%}
.T{opacity=80%}
```

This changes the final opacity of the placed object.

5.5.5 Soft Edges

```
.T{s=12%}
.T{soft=[12% 4%]}
.T{s=[5% 10% 15% 20%]}
```

`soft` creates a soft fade towards the edges of the object.

Accepted forms:

- one value: same fade on all sides
- two values: `[horizontal vertical]`
- four values: `[top right bottom left]`

Values are percentages.

5.5.6 Filter Copy

```
.T{f=myFilter}
.T{filter=myFilter}
.T{f=image1-9-1}
```

`filter` applies an existing SVG filter to the final placed object.

Accepted references:

- the id of a `<filter>` element, for example `f=myFilter`
- the id of another element that already has a filter applied, for example `f=image1-9-1`

When an element id is used, PnPInk copies that element's current `filter` reference. This is useful for reusing Inkscape-made effects such as color shifts, brightness tweaks or glows without rewriting the filter in the DSL.

5.5.7 Typical Use

```
photo.T{o=85%}
photo.T{s=10%}
photo.T{f=myWarmTint}
photo.T{f=image1-9-1 s=7%}
photo.T{o=75% s=[8% 3%]}
```

This is useful for:

- fading photos or textures
- softening cutout edges
- blending placed art into a tile or card background

5.5.8 Related Pages

- Fit-Anchor
- Source

5.6 Layout

Defines how a group of elements are arranged in a grid (e.g. cards on pages). Use this module when you want deterministic pagination and slot planning.

5.6.1 Syntax

```
Layout{ p=nxm g=[x y] o=[w1 h1 w2 h2] shape=... } or L{nxm g=... o=... s=...}
```

Can be applied to any object:

```
rect.L{}
```

```
[rect1 rect2 ...].L{}
```

```
{A4}.L{p=5x8 s=hexgrid}
```

5.6.2 Pattern (p=)

`p` is the structural core of layout. It defines how many slots exist per page before page breaks occur.

`p=` (or positional token) defines columns x rows.

```
p=3x2
p=0x0
p=3x?
p=?x4
p=-?x-?
```

`0x0` means auto-fit based on available area. `?` is also accepted as auto, so `0` and `?` are equivalent in pattern.

Order and Flips

Order and flips matter when instance numbering and print order must follow a specific physical workflow.

The grid token supports modifiers:

- `^` switches order to top-to-bottom, then left-to-right.
- `|` flips the axis (use after the number you want to flip).
- Negative numbers also flip that axis.

Examples:

```
p=3x2        -> left-to-right, top-to-bottom
p=3x2^       -> top-to-bottom first
p=3|x2       -> flip columns
p=3x-2       -> flip rows
p=-3x-2      -> flip both axes
```

5.6.3 Gaps (g=)

Use gaps to control spacing between slots without changing card/template size.

`g=` (or `gaps=`) defines spacing between slots.

```
g=[x y]
g=2
g=?
```

Rules:

- 1 value means `x=y`.
- 2 values mean `x` (horizontal) and `y` (vertical).
- Units and percentages are allowed.
- `?` means auto gap (distribute remaining space evenly to fit the final content area on that axis).

Examples:

```
g=2
g=[2 3]
g=[1% 3%]
g=[2 ?]
g=[? ?]   # same as g=[?] and g=?
```

Percentages require a known card size (shape preset or template size).

5.6.4 Offset (o=)

Offsets are useful for staggered layouts, including hex-like placement. They modify slot position patterns, not slot dimensions.

`o=` (or `offset=`) defines staggered offsets for alternating slots.

```
o=[w1 h1 w2 h2]
```

Notes:

- If only `w1 h1` are provided, `w2 h2` defaults to `-w1 -h1`.
- Offsets are applied after slot sizing and before final placement.

This is how hex-like or staggered grids are achieved without changing card size.

5.6.5 Shape (s=)

Shape presets provide normalized card/tile sizes so layouts remain consistent across projects.

Defines the shape preset of each grid cell.

Examples:

```
s=poker
s=minieuro
s=55.3x77.1
s=rect<55.3x77.1>
s=hex<24x33>
s=polygon<[5 23x32]>
```

Append `^` to swap width and height for a shape size or preset:

```
s=creditcard   -> 85.6 x 54
s=creditcard^  -> 54 x 85.6
```

See the Presets page for the full list of named sizes.

Hex Shapes

Hex shapes are specialized layout modes for maps and tile production. They automate spacing behavior that would otherwise require manual offset tuning.

Smart shapes adjust gaps and offsets without changing card size:

- `s=hexgrid` for maps, boards, overlays.
- `s=hextiles` for cuttable tiles with shared edges.

Inkscape tip: use `Tools > Stars and Polygons` (Shift+F9), set corners to 6, and hold Ctrl while resizing to keep alignment.

For practical examples and the differences between `hexgrid`, `hextiles`, and hex cut lines, see Hexes.

5.7 Hexes

PnPInk includes dedicated support for hex-based layouts.

This is useful for two common workflows:

- `hexgrid` for overlays, maps, and boards
- `hextiles` for printable hex tiles that must align edge to edge

Hex workflows often use two different line systems:

- `Marks` for global page/layout guides
- `Paths` for local lines attached to individual placed hexes

5.7.1 Syntax

Hex layouts are enabled from `Layout` using the shape field:

```
{A4}.L{p=6x5 s=hexgrid}
{A4}.L{p=6x5 s=hextiles}
```

You can combine them with normal layout options such as `p=`, `g=` and `o=`.

5.7.2 Hexgrid

`s=hexgrid` creates a staggered hex-style layout without changing the size of your source object.

Typical uses:

- map overlays
- area-control boards
- hex coordinate guides
- transparent helper layers placed over another page

PnPInk automatically applies the stagger needed for a practical hex arrangement, so you do not have to calculate the offsets by hand.

Example:

```
{A3^ @2}.L{s=hexgrid} 300
```

5.7.3 Hextiles

`s=hextiles` is meant for physical tiles.

Use it when each placed item is a real hex tile and adjacent tiles should share the same geometric structure page-wide.

Typical uses:

- modular terrain
- map tiles
- printable counters or region tiles

Compared with `hexgrid`, `hextiles` is oriented to production: neighboring tiles line up as a tile field, and marks can follow the real hex edges.

Example:

```
{A4}.L{p=5x6 s=hextiles}
```

5.7.4 Cut Lines In and Out of the Shape

When hex tiles are intended for printing and cutting, the most useful companion is `Marks{}`.

Marks can draw line segments on both sides of the tile edge:

- an **outer** segment, outside the tile
- an **inner** segment, inside the tile

This is controlled with `len=`.

```
.M{len=3}
.M{len=[3 2]}
```

Meaning:

- `len=3` means an outer segment of `3` and no inner segment
- `len=[3 2]` means outer `3`, inner `2`

For hextiles, these lines follow the real hex edges instead of behaving like simple rectangular crop marks.

This makes it easier to produce:

- shared cutting guides
- edge-aligned tile sheets
- visible internal guide lines when needed

You can also combine this with distance and border adjustments:

```
{A4}.L{p=5x6 s=hextiles}.M{len=[3 2] d=2 b=0}
```

In practice:

- `d=` controls how far the outer line starts from the edge
- `b=` shifts the line relative to the edge
- `len=[out in]` controls how much line appears outside and inside the shape

5.7.5 Marks vs Paths

These two modules solve different problems.

Marks

`Marks` is defined globally in the first column together with `Page{}` and `Layout{}`.

It follows the placed slots of the page and is mainly intended for cutting and registration.

Example:

```
{A4}.L{p=5x6 s=hextiles}.M{len=[3 2] d=2}
```

Paths

`Paths` is attached to the content of a specific cell.

It draws local geometry on that placed hex: edges, center lines, vertex connections, or links to neighboring hexes.

Example:

```
tile_id.P{path_style [a 5c 79]}
```

See Paths for the complete token system.

5.7.6 Recommended Usage

- Use `hexgrid` when the hex pattern is mainly a placement system.
- Use `hextiles` when the hex shape itself is part of the printable result.

5.7.7 Related Pages

- Layout
- Marks
- Paths

5.8 Paths

`Paths` generates helper paths directly on top of a placed shape.

Unlike `Marks`, which are defined globally in the first column and follow the page layout, `Paths` are attached to individual cell content. They are useful when each tile, hex, or placed object needs its own internal guides.

5.8.1 Typical Use

Use `Paths` when you want lines such as:

- a single hex edge
- a line from the center to one side
- a line between two vertices
- a curved connection between two sides
- a bridge from one hex center to the neighboring hex center through a shared side
- a chained route that continues through several neighboring hexes

This is especially useful for hex maps, hextiles, movement guides, borders, and local overlays.

5.8.2 Syntax

`Paths` is attached to the object token itself:

```
tile_id.P{path_style [a b]}
tile_id.P{path_style [5a 79]}
tile_id.P{path_style [ab cd]}
.P{path_style [a b d]}
```

You can define several styled groups in the same block:

```
tile_id.P{path_main [a c e] path_aux [5a 5c]}
tile_id.P{t=path_main [ab] t=path_aux [5A5]}
```

Each pair is:

- a style id
- followed by a token list in `[...]`

The style id may reference:

- one path
- or a group of paths, used as a style stack in the same z-order

5.8.3 Where It Applies

`Paths` belongs to the cell content, not to the page layout header.

You can attach it explicitly to a target:

```
tile_id.P{path_style [ab]}
```

or use it directly on the current placed target:

```
.P{path_style [ab]}
```

So this is the right mental model:

- `Marks` -> page/layout-level cutting or registration guides
- `Paths` -> per-cell geometry drawn on a placed target

5.8.4 Hex Nomenclature

For hex shapes, PnPInk uses:

- sides: `a b c d e f`
- vertices: `8 9 3 2 1 7`
- center: `5`

The vertices follow the same keypad-style convention already used elsewhere in the DSL.

5.8.5 Sides

Each lowercase letter refers to one hex side:

```
[a]
[b]
[c]
```

These draw the corresponding edge itself.

5.8.6 Center to Side

`5a`, `5b`, `5c` ... draw a straight line from the center of the hex to the midpoint of that side.

Examples:

```
[5a]
[5c 5f]
```

5.8.7 Vertex to Vertex

Two keypad numbers draw a straight segment between vertices.

Examples:

```
[79]
[93]
[12]
```

5.8.8 Side to Side

Two lowercase side letters draw a connection between side midpoints.

Examples:

```
[ab]
[ac]
[ad]
```

Depending on the relative position, this becomes:

- a straight line for opposite sides
- or a curved connection for non-opposite sides

5.8.9 Center to Neighboring Hex

An uppercase side letter in the form `5A5`, `5B5`, ... means:

- start at the center of this hex
- go through the midpoint of that side
- continue to the center of the neighboring hex on that side

Examples:

```
[5A5]
[5C5 5F5]
```

This is useful for:

- adjacency guides
- movement links
- map connectivity

5.8.10 Chained Neighbor References

You can also build continuous routes through several neighboring hexes.

Examples:

```
[5A9]
[3C3]
[5ABB3C2]
```

This means:

- `5A9`: from the current center to vertex `9` of the neighboring hex on side `A`
- `3C3`: from vertex `3` of the current hex to vertex `3` of the neighboring hex on side `C`
- `5ABB3C2`: one continuous polyline through several steps

If the chain is written without spaces, it stays as one continuous path. If you separate the tokens with spaces, PnPInk creates separate paths.

Important: in chained references, each new step starts from the last resolved hex, not from the original hex.

So:

```
[5ABB3C2]
```

continues from the hex reached by `ABB3`, and then applies `C2` from there.

5.8.11 Style Reuse

The path geometry comes from the tokens, but the visual appearance comes from the referenced style element.

So a common pattern is:

```
tile_id.P{path_thin [a b c] path_bold [5A5]}
```

This lets you draw different local guides with different strokes while keeping the syntax compact.

If the style id points to a group, PnPInk generates one path per child path in that group and preserves their stacking order. This is useful for multi-stroke styles such as railways, roads, or layered local guides.

5.8.12 Related Pages

- Hexes
- Layout
- Marks

5.9 Page

Defines page format, margins, and the global page cursor. Use `Page{}` whenever print format, orientation, or pagination position must change.

5.9.1 Syntax

```
Page{ size landscape border at }   or   P{A4}   or   {A4}
```

When `Page{}` appears in the first column, the keyword `Page/P` can be omitted. A bare `{}` means "continue using the previous page size".

Multipliers are allowed:

```
{3*A4} -> three A4 pages
{3}    -> shorthand for three pages
```

5.9.2 Size (default)

Page size is usually set once and then reused by state.

The default parameter is the page size:

```
Page{A4}
Page{Letter}
Page{23.3x34.45}
```

5.9.3 Landscape / Portrait

Orientation is part of page state and affects all following placements until changed.

```
Page{A4^}        -> landscape
Page{landscape}
Page{portrait}
```

5.9.4 Border (b=)

`b` defines the usable inner area (or outward expansion) for layout planning.

`b=` defines padding or margin around the page.

```
b=[-2]      -> 2 mm inward margin
b=[2 3 4 5] -> top, right, bottom, left
```

Percentages and absolute `WxH` values are allowed (same grammar as Fit).

For `%` borders in 1/2-token forms, percentages define absolute target scale:

- `b=[50%]` -> final size x0.5
- `b=[125%]` -> final size x1.25
- `b=[50% %]` -> height x0.5, width x1.0
- `b=[23x?]` / `b=[?x12]` are valid in `WxH` absolute-size mode (`?` = 100% on that axis).

5.9.5 Page Cursor (at / a / @)

Cursor control is for advanced pagination workflows, such as merging sections or forcing output positions.

`at` moves the global page cursor.

Accepted forms:

```
at=+3
a=-1
@5
A4@+2
{ @-1 }
```

Rules:

- `+n` / `-n` -> relative move.
- `n` -> absolute 1-based page (n -> index n-1).
- If the current page already has content, the engine jumps to a new page first.

5.10 Source

```
@{...} creates a source object from an external resource (file, icon, or URL), then uses it as a renderable object in the document.
```

Use Source when content comes from outside the template and must be resolved at generation time.

5.10.1 Syntax

All three forms below are equivalent entry points into the same source resolver pipeline.

```
Source{source_ref} or S{source_ref} or @{source_ref}
```

The `source_ref` can be:

- a path to a local file,
- an iconify reference,
- a web URL,
- or a virtual source (Wikimedia Commons, Pixabay, Openclipart, PnPInk Assets).

5.10.2 Local File Sources

Use local files for stable, reproducible builds where assets are versioned with the project.

```
@{ relative/or/absolute/path.png }
@{ C:\path\to\image.png }   # Windows
@{ ~/images/token.png }     # Linux/macOS
```

Equivalent local-source forms accepted in dataset cells:

```
Source{path/file.ext}
@{path/file.ext}
@{file.ext}
@file.ext
file.ext
```

Notes:

- `file.ext` is accepted only for known source extensions (`png`, `jpg`, `jpeg`, `gif`, `bmp`, `webp`, `svg`, `svgz`, `pdf`, `tif`, `tiff`).
- `file.ext}` is **not** valid syntax. The closing `}` only exists in braced forms:
- `@{file.ext}`
- `Source{file.ext}`

When no path is provided (`file.ext`), lookup order is:

1. Same folder as the SVG.
2. `assets/`
3. `images/`
4. `img/`
5. `imgs/` (compatibility)

Local sources support environment/home expansion:

- Windows: `%USERPROFILE%`
- Linux/macOS: `$HOME`, `${HOME}`, `~`

Inline text tokens

Inline icons can also use local shorthand directly inside text:

```
:bola.png:
```

Behavior:

- If the file is found by the same lookup rules above, it is treated as a source token.
- If not found, the token is kept as normal literal text (`:bola.png:`).

5.10.3 SVG Node Import (Optional)

For SVG sources only, you can target a specific node with `#id` :

```
@{ assets/icons.svg#heart_group }
@{ https://example.com/icons.svg#token_1 }
```

Behavior:

- without `#id` : the whole SVG is linked as a final image source,
- with `#id` : that node is imported and embedded (including required defs/references).

5.10.4 Iconify Sources

Use icon sources for semantic symbols (costs, resources, status icons) without managing local files manually.

```
@{ icon://icon_set/icon_name }
```

Examples:

```
@{ icon://noto/heart-suit }
@{ icon://mdi/account }
@{ icon://cat }   # uses default set (noto)
```

There is a default snippet definition mapping to the `noto` icon set:

```
:Ic(icon) -> @{ icon://noto/icon }
```

Notes:

- Icons are cached as SVG symbols in `<defs>` .
- If `iconify.py` is not available, a placeholder is created.
- To force a re-download, remove the symbol from `Object > Symbols` (Shift+Ctrl+Y).

5.10.5 Web Sources (HTTP/HTTPS)

Use web sources when assets are remote and can be cached at generation time.

```
@{ https://... }
@{ http://... }
```

Web sources are **downloaded and cached** into the `assets` folder next to the SVG (or project root). If the download fails, a placeholder symbol is created.

5.10.6 Virtual Sources

Virtual sources map high-level search expressions to real downloadable URLs. They are useful for exploratory workflows and rapid prototyping.

PnPInk supports virtual sources that resolve to real URLs:

```
@{ wkmc://query/size }
@{ pxby://query/size }
@{ oclp://query/size }
@{ pnp://asset_path }
```

PnPInk also supports dedicated map sources:

```
@{ osm://[...] }
@{ ofm://[...] }
@{ osm://madrid }
@{ ofm://spain/z4 }
```

These sources generate maps from simple URLs. See Maps for the map syntax and examples.

Placed sources can also be adjusted afterwards with Transform, for example to reduce opacity or soften edges.

- `wkmc://` (Wikimedia Commons) supports:
- search text: `wkmc://query/size`
- specific file: `wkmc://File:Name.ext/size`
- category files: `wkmc://Category:Name/size`
- `pxby://` (Pixabay) supports:
- normal search list: `pxby://query/size`
- direct lookup for no-space query when possible (single hit / id).
- `oclp://` (Openclipart) supports:
- normal search list: `oclp://query/size`
- specific image by id/detail: `oclp://id:12345/size` (resolved internally via `https://openclipart.org/detail/12345/...`)
- `pnp://` (PnPInk Assets) supports:
- direct asset path: `pnp://birds/egg1`
- default `.png` extension when omitted
- friendly numeric fallback: `pnp://birds/egg` -> first matching numbered asset such as `egg1.png`
- optional global lookup when the name is unambiguous: `pnp://egg`

Size accepts:

- presets: `tiny`, `small`, `medium`, `large`, `xlarge`, `largest`
- vector mode: `svg` (when available from the provider)
- minimum side: `N` (example: `1000`)
- minimum width+height: `WxH` (example: `1000x1000`)

Examples:

```
@{ wkmc://"The Ancestral Homes of Britain"/large }
@{ wkmc://"File:Complete_Saxonian_deck.jpg"/1000 }
@{ wkmc://"Category:Complete_decks_of_playing_cards_laid_out"/1000x1000 }
@{ pxby://castle/1200 }
@{ oclp://wolf/large }
@{ oclp://id:24829/largest }
@{ pnp://birds/egg }
@{ pnp://IA/icons/crown1 }
```

Multiple-result virtual sources can be selected outside the source with a 1-based selector:

```
@{ wkmc://"The Ancestral Homes of Britain"/medium }[2 4..12 15..26]
*@{ pxby://castle/large }[1..20]
```

Selector notes:

- indices are 1-based,
- out-of-range indices are ignored with warning,
- without selector, if multiple results exist, the first one is used (warning in log).
- for `wkmc://`, category/search results keep the API order after size filtering; PnPInk does not reorder them by image dimensions.

Wikimedia Commons category catalogs:

- each `wkmc://Category:...` source found in the dataset creates an internal 0-based catalog variable:
- first category: `_wkmcc1`
- second category: `_wkmcc2`
- each item exposes the useful fields returned by Commons:
- `title`
- `url`
- `thumburl`

```
${_wkmcc1[0].title}
${_wkmcc1[0].url}
${_wkmcc1[0].thumburl}
@{ wkmc://${_wkmcc1[0].title}/1000 }
@{ ${_wkmcc1[0].thumburl} }
```

`pnp://` PnPInk Assets

`pnp://` resolves against the public `pnpink-assets` index generated on push. It is meant for small reusable art pieces that can be placed automatically from a dataset without downloading them into every project.

Resolution rules are intentionally simple:

- if you provide a full asset stem, it is used directly,
- if you omit the extension, `.png` is assumed,
- if you omit the numeric suffix, PnPInk tries the lowest matching numbered asset,
- if you omit the folder, PnPInk tries a global lookup and warns if the name is ambiguous.

Examples:

```
@{ pnp://egg }            # may resolve to birds/egg1.png
@{ pnp://birds/egg }      # resolves within the birds collection
@{ pnp://IA/icons/crown1 }
```

The source behaves like any other image source, so it can be combined with Fit-Anchor and Transform:

```
@{ pnp://egg }~i7
@{ pnp://birds/egg3 }.T{o=80%}~[110%]8!
```

5.10.7 Fit and Placement Behavior

After resolution, a source behaves like any other placeable target in Fit/Anchor terms.

A Source is treated as a final placeable object, so you can apply Fit/Anchor operations exactly as with other objects.

5.10.8 Internal Model

This internal model is important for performance and file size on large projects.

Sources are instantiated as clones of symbols (`<use>`), not inline geometry copies. This keeps the SVG compact and avoids repeating heavy geometry.

5.10.9 Spritesheets (@)

Spritesheets allow one asset to provide many addressable frames. This is useful for token sheets, icon atlases, and catalog-like image packs.

The **Layout** module can also be applied to composite sources, such as spritesheets.

Example definition inside a comment block (before the dataset):

```
# @sp1 = @{sheet.png}.Layout{p=3x2^ s=poker g=[4 3]}
```

This creates a spritesheet from `sheet.png` with a 3x2 grid of poker-sized cards, spaced 4 mm horizontally and 3 mm vertically.

Then, you can reference any frame in the dataset as:

```
@sp1[14] -> frame 14 (third page, second row)
@sp1[2][1][3] -> page 2, column 1, row 3
@sp1[1..6] -> range selector
@sp1[1 4 7] -> explicit list selector
```

(`^` in the grid reverses numbering direction: first rows, then columns.)

This spritesheet can be used as a source in any dataset, positioned with Fit parameters (e.g. `~i6`) as any other target.

Notes:

- Alias definitions are read from comment lines before dataset rows.
- Frame selectors are 1-based.
- If a frame is out of range, a warning is logged and placement is skipped.

5.11 Maps

PnPInk can generate maps directly from simple source URLs such as `osm://...` and `ofm://...`.

PnPInk resolves the selected area automatically, downloads the needed vector tiles, builds a light layered SVG, and inserts it directly into the document. The result remains compatible with Fit/Anchor and is easy to edit afterwards because it is regular SVG content inside the page.

5.11.1 Syntax

All map sources use one of these forms:

```
@{ osm://[lat1 lon1 lat2 lon2] }
@{ ofm://[lat1 lon1 lat2 lon2] }

@{ osm://madrid }
@{ ofm://spain }

@{ osm://spain/z4 }
@{ ofm://madrid/z8 }
```

- `osm://[...]` and `ofm://[...]`
- use a bounding box defined by two opposite corners
- `osm://place` and `ofm://place`
- use a place name
- `/zN`
- forces a specific zoom level

If no zoom is forced, PnPInk chooses it automatically.

As a simple rule:

- use `osm://` for a lighter base map
- use `ofm://` for a richer map with more terrain and landcover detail

5.11.2 Coordinates

Bounding boxes use this order:

```
[lat1 lon1 lat2 lon2]
```

The two points are opposite corners of the rectangle.

You can obtain coordinates easily from:

- OpenStreetMap: Open the map, go to `Export`, and inspect the selected area
- Google Maps: right click on a point and copy the coordinates

5.11.3 What PnPInk Automates

You only provide the area or the place name.

PnPInk automatically:

- resolves the place when needed
- chooses an appropriate zoom if you do not force one
- downloads the necessary tiles
- joins adjacent tiles
- clips the result to the requested area
- inserts the generated SVG into the document

Downloaded tiles and place lookups are cached automatically in `assets/maptiles/`, so repeated renders of the same areas are much faster.

5.11.4 Map Content

The generated SVG is layered and grouped, so it can contain examples such as water, rivers, roads, landcover, parks, labels, places, and mountain peaks depending on provider and zoom.

This makes it practical to style or edit parts of the map later inside Inkscape.

5.12 Marks

The Marks module generates cut marks aligned to the **Layout grid slots** of the **main template** (first dataset column). Use it when output is intended for physical cutting and alignment consistency matters.

`Marks` is a page/layout-level system. If you want local lines attached to individual hexes or tiles, use Paths instead.

Marks are defined in the **first header cell** (same place as `Page{}` and `Layout{}`), for example:

```
{A4}.L{p=3x3 g=2}.M{ mk_style len=[3 2] d=2 }
```

5.12.1 Slot-Based Behavior

This behavior explains why marks remain correct even when pages, gaps, or offsets change.

Marks are generated **per slot** (per placed instance in the grid), using the transformed bounding box of the main template for that slot.

This means:

- Marks follow the real grid placement (including gaps, offsets, page breaks, rotations, etc.).
- Marks are not tied to a specific template column; they follow the main layout slots.

5.12.2 Syntax

The syntax is short, but each parameter controls a different geometric aspect of cut lines.

```
Marks{ style len distance border layer }
```

Shorthand:

```
M{ ... }
```

Elements inside `{}` are space-separated. Lists are written in `[]` and are space-separated.

5.12.3 Default Parameter (Style)

Style controls visual appearance, not mark geometry. It is resolved from existing SVG elements.

The default parameter of `Marks{}` is the style id. This means `style=` can be omitted:

```
.M{ mk_style len=3 d=2 }
```

is equivalent to:

```
.M{ s=mk_style len=3 d=2 }
```

Style Sources

A style id (s=) references an SVG element by ID:

- If the ID refers to a single element (path/shape), its stroke-related style is copied to the generated mark paths.
- If the ID refers to a group (`<g>`) containing multiple child paths, it is treated as a **style stack**. The mark geometry is generated multiple times, each copy receives the style of one child, and the result is layered.

If no `s` is provided, a default style from preferences is used.

5.12.4 Length (len=)

Length controls how far marks extend outside and inside slot edges.

`len=` defines the length of the mark segments.

Rules:

- Scalar: `len=3` means **external length = 3**, internal length = 0.
- Two-value list: `len=[out in]`.

If the layout uses offsets (staggered grids), a scalar `len=3` is treated as `[3 3]` to keep internal marks visible.

For hex-specific usage, including `hextiles` and lines inside/outside the shape, see Hexes.

5.12.5 Distance to Card (d=)

Distance is the gap between the card edge and the start of the external mark segment.

`d=` sets the distance between the card edge and the **external** mark origin.

Default: `d=2`.

`d` follows the same list grammar as border in Fit/Page:

```
d=[2 3 4 5] -> top, right, bottom, left
```

5.12.6 Border Pattern (b=)

Border offsets let you fine-tune mark placement against bleed/margin strategies.

`b=` defines the cut-mark border offsets using the same conventions as borders in Fit/Page.

Default: `b=0`.

5.12.7 Output Layer (layer=)

Separating marks into a dedicated layer helps review, export, and printer handoff.

`layer=` defines the target Inkscape layer where marks will be inserted.

Default: `layer=marks`.

If the layer does not exist, it is created.

Example:

```
.M{ layer=cutmarks }
```

5.12.8 Scope

This section documents current implementation limits to avoid false expectations.

The `scope=` parameter is documented but **not implemented** in the current engine. Marks are always generated on the front pass.

5.12.9 Examples

Basic cut marks with default style

```
{A4}.L{p=3x3 g=2}.M{}
```

Using a style element

```
{A4}.L{p=3x3}.M{ mk_cut }
```

Overlay style using a group

```
{A4}.L{p=3x3}.M{ mk_cut_stack len=[3 2] }
```

5.13 Gradients (Experimental)

> **Warning**
>
> This feature is currently **in development**. Syntax and behavior can still change in future versions.

5.13.1 Overview

PnPInk can create SVG `linearGradient` definitions directly from dataset comment directives. These gradients are generated in `<defs>` and can then be used from any SVG style, for example:

```
fill: url(#gradientX);
stroke: url(#gradientX);
```

5.13.2 Where to define gradients

Gradient definitions are read from dataset comment lines (same global/local comment mechanism used by snippets and spritesheets).

Typical location:

- top comment block of your dataset file,
- before data rows,
- usually with `#` at the start of the first cell.

Example:

```
# gradientX=G{b88326ff^25 [-25 12 5] [#BA8726FF 35 20 10]}
```

5.13.3 How to apply gradients

Once a gradient ID exists, apply it in your SVG element style:

```
style="fill:url(#gradientX);"
```

Or as stroke:

```
style="stroke:url(#gradientX);"
```

PnPInk does not require a special dataset field for this part. You use normal SVG styling with the generated gradient ID.

5.13.4 Syntax

Two forms are supported.

Short form (`G{...}`)

```
# gradientX=G{basecolor^angle [stop1] [stop2] ...}
```

Example:

```
# gradientX=G{b88326ff^25 [-25 12 5] [#BA8726FF 35 20 10]}
```

Meaning:

- `basecolor`: base color of the gradient (`#RRGGBB` or `#RRGGBBAA`, `#` optional).
- `^angle`: linear gradient angle in degrees (optional, default `0`).
- each stop block: `[...]` definition for local highlight/shadow behavior.

Long form (`Gradient{...}`)

```
# gradientX=Gradient{basecolor=b88326ff rotate=25 stops=[[-25 12 5] [55 35 20 10]]}
```

Parameters:

- `basecolor` (or `base` / `color`): required.
- `rotate` (or `angle` / `r`): optional, default `0`.
- `stops=[ ... ]`: required in long form.

5.13.5 Stop definition

Each stop can be numeric-light or explicit-color.

Numeric light stop

```
[light pos a b]
```

- `light`: `-100..100`
- positive -> lighter (towards white),
- negative -> darker (towards black).
- `pos`: position in `%`.
- `a`: left width in `%`.
- `b`: right width in `%` (optional, if missing then `b=a`).

Example:

```
[-25 12 5]
```

This creates a darker zone centered at `12%`, with left/right width `5%`.

Explicit color stop

```
[color pos a b]
```

- `color`: `#RRGGBB` or `#RRGGBBAA`.
- `pos`, `a`, `b`: same meaning as above.

Example:

```
[#BA8726FF 35 20 10]
```

This creates a custom-color peak at `35%`, with asymmetric widths (`20%` left, `10%` right).

5.13.6 Behavioral details

Current implementation rules:

1. Base color is present at `0%` and `100%`.
2. Each stop adds three internal points:
3. `(pos-a) -> base color`
4. `pos -> peak color`
5. `(pos+b) -> base color`
6. Offsets are clamped to `0..100%`.
7. If multiple points collapse to the same offset, last generated value wins.
8. `light` values are converted by mixing base color with white/black.

5.13.7 End-to-end example

Dataset comments:

```
# gradientGold=G{b88326ff^25 [-25 12 5] [55 35 20 10]}
```

SVG object:

```
<rect id="card_bg" style="fill:url(#gradientGold);stroke:#000;stroke-width:0.3" />
```

Result:

- `gradientGold` is created in `<defs>`,
- object fill resolves to the generated linear gradient.

5.13.8 Current limitations

- Only `linearGradient` is generated.
- `radialGradient` and mesh gradients are not part of this feature yet.
- Because this is experimental, parser details and interpolation behavior can evolve.

5.14 Advanced

Advanced DSL features that are implemented in code and useful in larger projects.

5.14.1 Alias Definitions

Aliases reduce repetition in large datasets and make rows easier to read.

You can define aliases and reuse them later:

```
@hero = @{assets/hero.png}
@icons = [icon_hp icon_atk icon_def]
```

Then reference by index:

```
@hero
@icons[2]
@icons[1..3]
@icons[*]
@icons[1 4 7]
```

5.14.2 Source Suffixes

Suffixes are useful when source resolution and fit behavior must be expressed in one token.

Source expressions support both long Fit and shorthand ops:

```
@{assets/token.svg}.Fit{mode=i anchor=5}
@{assets/token.svg}~i5^15|
```

This applies the same Fit/Anchor logic used for normal SVG IDs.

5.14.3 Page Cursor Control (`at`)

Use cursor control when output must start at a specific page position.

`Page{}` supports page cursor movement with `at` / `a` / `@`:

```
{A4 @+3}
Page{A4 at=-1}
Page{A4^@5}
```

This controls where the next generated content starts in the global page sequence.

5.14.4 Page Break Blocks

These forms are useful for explicit pagination control between logical sections.

Standalone page blocks are valid and useful:

```
{}      # break to next page
{3}     # advance by 3 pages
{3*A4}  # multiplier + size
```

5.14.5 Leading Cell Composition

Combining directives in column A allows row-level orchestration without extra columns.

Column A data rows can combine multiple directives in one cell:

```
{A4 b=[-5]} .L{p=3x3 g=2} .M{mk_default d=2} [10 3- 5]
```

Supported order is flexible, but the parser expects:

1. Optional `Page{}` block.
2. Optional `.M{}` marks block.
3. Optional `.L{}` layout block.
4. Optional copy/hole tail (`[10 3- 5]` or trailing number).

5.14.6 Inline Icon Tokens in Text

These tokens allow icon insertion directly in flowing text while preserving typographic behavior.

Text rendering supports inline icon tokens:

```
:heart_icon:
:@{icon://noto/heart-suit}:
:@{icon://noto/star}~i5^10:
```

Token forms implemented:

- `:id:` uses an existing SVG id.
- `:@{...}:` uses a Source token with optional fit suffix.
- `:S{...}:` and `:Source{...}:` are equivalent source forms.

6 PnPInk User Manual

6.1 Introduction

PnPInk is a **free, open-source, and cross-platform toolkit** designed to create high-quality *print-and-play* materials for board games -- cards, boards, tokens, player aids, and more -- directly inside **Inkscape**.

Unlike other generators, **PnPInk is not a separate application but a native extension suite that transforms Inkscape into a complete publishing environment -- capable of accompanying creators from the very first prototype designs to the final, fully professional print composition.**

6.2 What PnPInk Does

PnPInk automates the creation of:

- **Card decks** of any size or format.
- **Boards and punchboards**, including **hexagonal or square grids**.
- **Player sheets**, tiles, counters, and rule inserts.

It combines **data-driven generation** (from Google Sheets or CSV files) with **Inkscape's full graphic power** -- gradients, filters, clipping masks, paths, symbols, and layers -- to produce editable, print-ready layouts with pixel-perfect precision.

6.3 Key Principles

Principle	Description
Open Source	PnPInk is released under an open license. All files remain accessible, editable, and transparent.
Cross-Platform	Works wherever Inkscape runs: Windows, macOS, or Linux.
Non-Destructive	Everything you generate stays as pure SVG -- fully editable afterwards.
Professional Output	Exports **PDFs with CMYK color**, ICC profiles, and print standards (FOGRA, PDF/X).
Accessible & Visual	No programming required -- all actions are done through menus, dialogs, and clear visual feedback.

6.4 Power of Inkscape Integration

Because PnPInk is built *inside* Inkscape, it inherits all of its advanced features:

- **Full vector control:** Bezier paths, shapes, and text on path.
- **Complex styling:** gradients, blurs, blend modes, and filters.
- **Image management:** linked or embedded images, with live previews.
- **Multi-page support:** each layout can produce complete, paginated projects.
- **Universal format compatibility:** SVG, PDF, PNG, EPS, DXF, and more.

This makes PnPInk suitable for both quick prototypes and professional offset printing.

6.5 Intelligent Data and Automation

PnPInk can read structured data directly from:

- **Google Sheets** (live connection with authentication), or
- **CSV files** exported from any spreadsheet program.

Each row in the data corresponds to a card, tile, or component.\ PnPInk replaces variables automatically -- names, values, icons, or images -- to generate all items in a single command.

You can also control elements through short, **intuitive syntax rules**:

- define page size and margins (`{A4 b=[5 4]}`),
- define the grid (`.L{p=3x3 g=2}`),
- position elements with anchors (`ID~i8` or `ID.Fit{a=8}`).

This nomenclature is **compact, logical, and extremely powerful**, allowing advanced layouts without complex scripting.

6.6 Extensible by Design

PnPInk includes an **internal Python engine** that lets power users create custom behaviors, filters, or generation logic.\ At the same time, the system remains compatible with **external programming** through Google Sheets formulas, scripts, and data manipulation -- perfect for dynamic prototypes or large datasets.

Together, these layers make PnPInk both **simple for beginners** and **limitless for experts**.

6.7 Output Quality and Professional Printing

All final projects can be exported as:

- Editable SVG files for on-screen review.
- **Print-ready PDFs** with CMYK color conversion and **bleeds, margins, and crop marks**.
- Optional **PDF/X compliance** for professional pre-press workflows.

This ensures consistent color reproduction and compatibility with professional printers and services.

6.8 Typical Use Cases

- Design and print your own card game in minutes.
- Generate hundreds of cards from a single Google Sheet.
- Build modular or hexagonal boards with automatic alignment.
- Create color-managed PDFs for offset printing.
- Share open, editable project files with collaborators worldwide.

6.9 Why PnPInk

PnPInk bridges the gap between **creativity** and **production**.\ It combines the artistic freedom of Inkscape with the precision and automation of modern data tools -- delivering a unique environment where *design, logic, and craft meet*.

6.10 PnPInk Basic Workflow (User Guide)

6.10.1 What you need

- **Inkscape** (Windows, macOS, or Linux).
- **PnPInk extension** installed.\ After installing, **new menus** appear in Inkscape under `Extensions > PnPInk DEV` (e.g. `DeckMaker v0.24dev`).

6.10.2 Your first project: the quick overview

1. **Design the template** in Inkscape\ Create a single "master" card (or board). This can be a full finished design, or a **generic frame** with placeholders (title, number, icon, art frame, background, etc.). This will be your **TEMPLATE**.
2. **Name things clearly** ** **Give each element a meaningful** ID **(e.g., title, cost, icon, art, bg). IDs let PnPInk know** what to replace** later.
3. **Prepare your data** ** **Use** Google Sheets **or a** CSV **file. Each** row **represents one card (and can include "copies" or quantity). Add columns that match your IDs--plus any other properties you want to control (texts, colors, styles, images, counts, etc.). There's** no limit** to how many you define.
4. **Choose a layout** ** **Tell PnPInk how to arrange cards on pages (paper size, grid, margins, gaps). Layouts are written with a** short, intuitive notation** (e.g., `{A4 b=[5 4]}` with `.L{p=3x3 g=2}`).
5. **Generate** ** **PnPInk clones the template for each row, replaces variables, positions/adjusts elements with** Anchor & Fit**, arranges them into** multi-page **documents, and prepares** print-ready PDFs **(including** CMYK/ICC** for professional printing).

6.10.3 IDs: how elements are matched to data

- **What they are:** IDs are the "names" of objects in your template (texts, shapes, groups, images).
- **How to use them:** If your CSV/Sheet has a column **title**, and your template has a text object with ID **title**, PnPInk will insert that row's text there.
- **Best practices:**
- Keep IDs **short and descriptive** (title, subtitle, icon, art, frame, bg).
- One purpose per ID (avoid duplicates unless you intend to repeat the same value).
- Use **groups** with an ID when multiple items belong together (e.g., a framed art block).

6.10.4 Layouts & pagination (short notation)

- **What layouts do:** They control paper size, card size/quantity per page, margins, and gaps.
- **Notation examples:**
- `{A4}` with `.L{p=3x3}` -> A4 paper, 3 columns x 3 rows.
- `{A4 b=[5 4]}` with `.L{p=3x3 g=2}` -> margins and gaps.
- Orientation toggles and presets let you switch between portrait/landscape and standard card formats.
- **Why it matters:** One template can produce **hundreds of cards**, arranged automatically into **clean, paginated sheets**--ready for cutting or die-cut.

6.10.5 Anchor & Fit (precise placement without manual nudging)

- **Anchor** = where the element aligns inside its target area (e.g., 8 = top-center, 5 = center, 3 = bottom-right).
- **Fit** = how it scales to **fill or respect** the frame:
- none (no scaling),

- contain (keep ratio, fit inside),
- cover (keep ratio, fill completely, may crop),
- stretchXY (scale independently).
- **Margins and bleed:** You can add **margins or negative margins** (e.g., [2] = 2mm padding on all sides; [-1] = extend 1mm beyond for bleed).
- **Result:** Titles center perfectly, icons snap to corners, images fill art frames, and backgrounds expand to edges--**consistently** across every card.

6.10.6 Inline icons (type the name, get the icon)

- **What it is:** Write icon names directly in text (e.g., :sword:, :lightning:) and PnPInk replaces them with **vector icons**--styled to match your fonts and colors.
- **Scale & style:** Icons live **inside the text flow**, so they line up with your typography and inherit styles.
- **Big libraries:** PnPInk can connect to **large online icon sets (200k+ icons)** that you can **call by name**. No manual importing--just type the name.
- **Use cases:** Resource symbols, status icons, action markers, rarity pips--**instantly consistent** across the whole project.

6.10.7 Snippets & variables (write once, reuse everywhere)

- **Variables:** In text fields, you can insert **placeholders** tied to your data (names, numbers, colors, costs, etc.). When you generate, they're replaced automatically.
- **Snippets:** Think of them as **reusable text chunks** or micro-templates. Example use cases:
- A standardized **rules block** you reuse on multiple cards.
- A **cost line** that composes icons + numbers based on data.
- Language variants or conditional content (e.g., add a keyword only if a value exists).
- **Benefit:** You keep your template **clean and consistent**, even for complex sets.

6.10.8 Asset sources (images, art, symbols)

PnPInk can pull artwork and symbols from multiple places:

- **Local files** you link in your data (e.g., art=images/dragon_03.png).
- **Document symbols** (shared elements duplicated across cards).
- **Sprite sheets or multi-page packs** (see "wrappers" below).
- **Online icon/name catalogs** (call by name, automatically resolved).

Tip: Always keep a simple **asset folder** structure; name files clearly to match your rows.

6.10.9 Spritesheets and virtual sources (advanced, optional)

- **Spritesheets** let you cut a bitmap into a grid and reference frames by index.
- **Virtual sources** let you reference web catalogs (Wikimedia Commons, Pixabay, Openclipart) using `wkmc://`, `pxby://`, or `oclp://`, including `File:` and `Category:` modes on Wikimedia plus size targets like `large`, `1000`, or `1000x1000`.
- This is ideal for **large libraries** or external packs where manual importing would be tedious.

6.10.10 Style control without limits

- Because everything happens **inside Inkscape**, you have the full power of:

- **Gradients, filters, blends, masks**,
- **Layers and symbols**,
- **Precise vector paths** and **text styling**.
- PnPInk simply **automates** the boring part; you keep **artistic control** at all times.

6.10.11 Professional output (CMYK & print standards)

- Export **PDFs with CMYK color**, **ICC profiles**, and optional **PDF/X** compliance for offset printing.
- Add **bleed, crop marks, and safe margins** automatically.
- Keep the editable **SVGs** for future updates, translations, or expansions.

6.10.12 Typical end-to-end flow (checklist)

1. Open Inkscape -> confirm **PnPInk menus** are visible.
2. Create your **template** (card/board) and assign **clear IDs**.
3. Prepare your **Google Sheet or CSV**:
4. One **row per card** (or copies).
5. Columns named after your **IDs** + any extra properties (e.g., icon, art, rarity, color, bg, copies, layout, etc.).
6. Choose a **layout** (paper size, grid, margins).
7. Use **Anchor & Fit** for placement and scaling.
8. Write **inline icons** right inside text by **name**.
9. Use **snippets/variables** for reusable blocks and smart content.
10. (Optional) Connect to **online icon libraries** and **wrappers** for multi-page asset packs.
11. **Generate** -> review pages -> export **PDF (CMYK)** for print.

7 Basic Workflow

This chapter is a practical "first successful run" guide. If you are starting from zero, complete it once end-to-end before reading full DSL reference pages.

7.1 What you need

Before running DeckMaker, confirm these prerequisites:

- Inkscape installed.
- PnPInk extension installed and visible in the Inkscape menu.

7.2 Create a Template

This step defines the visual source that all generated instances will clone.

Draw one card (or tile) as a normal SVG group.

Recommended:

- Create a rect that defines the card size (this is the **template bbox**).
- Give IDs to all elements you want to drive from data.

Inkscape ID workflow:

See Introduction -> How IDs connect data to graphics.

7.3 Prepare the Dataset

This step connects table columns to SVG IDs so each row can produce one variation.

Create a CSV or Google Sheet where:

- Column A contains the template bbox id.
- Column B+ contains IDs that match your SVG objects.

If using Google Sheets, set `GSheet ID` and optional `Sheet!range/gid` in DeckMaker. Public mode uses numeric `gid` (empty means `gid=0`); private mode uses OAuth and is more secure. See Dataset Format and Sources.

Example:

```
card_bbox,title,cost,art
,Fireball,3,images/fireball.png
,Shield,2,images/shield.png
```

7.4 Define Page and Layout

This step controls where generated instances are placed and how many fit per page.

Put a page/layout preset in the first cell of column A:

```
{A4 b=[-5]} L{p=3x3 g=2}
```

You can also put it in a control-only row:

```
{A4 b=[-5]} L{p=3x3 g=2},,,
,Fireball,3,images/fireball.png
```

If columns B+ are empty in that row, no card is generated for that line.

This means:

- A4 page with a 5 mm inward margin.
- 3x3 grid with 2 mm gaps.

7.5 Run DeckMaker

This is the generation step: template + dataset + page/layout become output pages.

In Inkscape, run `Extensions > PnPInk DEV > DeckMaker` and select the dataset.

Result:

- PnPInk clones the template for each row.
- Applies Fit/Anchor and Sources.
- Creates pages and exports layouts ready for print.

7.6 Useful Inkscape Panels for PnPInk

Inkscape provides right-side dock panels that are very useful when working with PnPInk. The most relevant ones are:

- `Object > Layers and Objects...` (Shift+Ctrl+L): inspect object tree, layer/group hierarchy, and Z-order.
- `Object > Object Properties...` (Shift+Ctrl+O): edit object metadata and properties that affect matching and behavior.
- `Object > Symbols` (Shift+Ctrl+Y): inspect symbol definitions used by Source and icon workflows.
- `Object > Fill and Stroke` (Shift+Ctrl+F): review visual style values that can influence readability and print output.
- `Object > Transform...` (Shift+Ctrl+M): apply deterministic transforms before generation if geometry is inconsistent.
- `Object > Align and Distribute` (Shift+Ctrl+A): enforce consistent alignment in templates before cloning.

For ID-specific inspection/editing panels, use the single reference in Introduction -> How IDs connect data to graphics.

8 DeckMaker GUI

DeckMaker is the user-facing PnPInk window used to generate decks from an SVG template and a dataset.

The window title shows the running DeckMaker version and the current template file name. The full template path is not shown in the form because the extension is launched from the active Inkscape document.

8.1 Deck tab

The Deck tab contains the data source controls, the main actions, and the live activity log.

8.1.1 Data source

Field	Purpose
`GSheet ID`	Google Spreadsheet id. This is only needed for Google Sheet sources.
`Range / gid`	Optional sheet range or grid id. Leave empty when the default source selection is enough.
`Source`	Selects how the dataset is loaded. Supported values are empty/default, local CSV, Google Sheet OAuth, and Google Sheet public.

For local CSV workflows, the spreadsheet fields can be left empty when the source is resolved from the template or from the selected file.

8.1.2 Actions

Control	Purpose
Auto checkbox before `Generate`	Starts generation automatically when the GUI opens.
`Generate`	Builds the output SVG from the current template and dataset.
Auto checkbox before `Open SVG`	Opens the generated SVG automatically after generation.
`Open SVG`	Opens the generated SVG through the same Inkscape launch path used by the extension. It does not use the operating system default application.
Auto checkbox before `Export`	Runs export automatically after generation.
`Export`	Runs the export pipeline configured in the Export tab.

8.1.3 Progress and log

The log area shows the current generation stage and relevant export stages. During generation, the progress text uses `Generating records` and shows the achieved speed in `records/min` when enough timing data is available.

For debugging, the full log is written to `src/pnpink.log` next to the extension sources.

8.2 Export tab

The Export tab configures PDF, PDF/X, and additional file exports. See Export for the detailed export pipeline.

The most important controls are:

Control	Purpose
`PDF export`	Enables standard PDF export.
`PDF output profiles`	Selects one or more PDF profiles: `default`, `screen`, `ebook`, `printer`, `prepress`.
`PDF/X export (CMYK)`	Enables CMYK PDF/X export through Ghostscript.
`Raster filters`	Chooses how filtered SVG content is handled before PDF export.
`Other formats`	Exports page images or alternate formats such as PNG, JPEG, TIFF, WebP, PS, EPS, EMF, or WMF.
`DPI`	Output resolution used by Inkscape exports.
`JPEG quality`	JPEG quality for JPEG-based outputs.

8.3 Preferences tab

The Preferences tab currently exposes SVG output splitting:

Control	Purpose
`Split SVG`	Writes the generated output as multiple SVG parts.
`Part target MB`	Target size for each generated SVG part.

Splitting is useful for very large decks because Inkscape export can be faster and more stable with smaller SVG files. Very small part sizes create many chunks and can increase overhead.

8.4 Generated files

Generation writes an editable SVG output derived from the template name, usually using the `_output.svg` suffix.

When SVG splitting is enabled, the parts are written to a sibling chunks directory. Export reuses those existing chunks when available.

8.5 Advanced preferences

Most preferences are stored in `src/preferences.ini`. The GUI writes explicit changes immediately, but it should not rewrite all preferences just because the window is closed.

Useful advanced keys:

Key	Values	Purpose
`template_engine`	`legacy`, `composed`, `composed-instance`	Template instantiation engine. `legacy` is the normal engine. `composed` is experimental and intentionally fails on unsupported templates. `composed-instance` keeps composed clones anchored to the first generated instance instead of `<defs>`.
`inline_icons_bbox_backend`	`query_all`, `shell_per_text`	Inline icon bbox measurement backend. Use `query_all` for normal work. `shell_per_text` is kept for investigation only.
`inkscape_shell_workers`	integer >= 1	Parallel Inkscape shell workers used during export.
`split_svg_output`	`0`, `1`	Enables generated SVG parts.
`split_svg_chunk_mb`	integer >= 1	Target part size in MB.

8.6 Troubleshooting

If generation appears to use old settings, close the DeckMaker window before editing `preferences.ini`. The GUI should not save preferences on close, but explicit GUI interactions still save the corresponding preference.

If Inkscape hangs while opening a very large generated SVG, check whether the Export panel is being restored by the Inkscape profile. On Windows this state is stored in `%APPDATA%\inkscape\dialogs-state-ex.ini`; closing the Export panel or resetting that dialog state avoids the hang. CLI export and isolated-profile launches are not affected.

If inline icons are missing or misaligned, check `inline_icons_bbox_backend`. The supported backend is `query_all`. The `shell_per_text` backend is experimental because Inkscape shell has shown inconsistent bbox results for inline text spacers.

If the composed template engine fails with an "unsupported template" error, switch `template_engine` back to `legacy`.

9 Export

The export pipeline operates on the generated SVG output, not on the original template. The usual workflow is:

1. Generate the deck SVG.
2. Optionally split the generated SVG into smaller parts.
3. Export PDF, PDF/X, or another format from the generated SVG or its parts.

9.1 Standard PDF export

Enable `PDF export` in the Export tab to create standard PDF output.

You can select one or more profiles:

Profile	Typical use
`default`	General-purpose PDF output.
`screen`	Lower weight output for screen review.
`ebook`	Compact output for digital distribution.
`printer`	Print-oriented PDF.
`prepress`	Higher quality print/prepress-oriented PDF.

The selected profiles are merged and normalized through Ghostscript after Inkscape has produced temporary page or chunk PDFs.

9.2 PDF/X CMYK export

Enable `PDF/X export (CMYK)` when the print provider requires a PDF/X file and CMYK conversion.

Controls:

Control	Purpose
`ICC`	Output CMYK ICC profile. Built-in profile names and absolute `.icc` or `.icm` paths are supported.
`PDF/X`	Selects `PDF/X-1a`, `PDF/X-3`, or `PDF/X-4`.
`Text in pure black`	Preserves text as pure black when supported by the Ghostscript conversion path.

PDF/X export requires Ghostscript. If the selected ICC profile cannot be resolved, the exporter falls back to the Ghostscript default CMYK profile and writes a warning to the log.

9.3 Raster filters

Filtered SVG content can be expensive or unreliable when exported directly. `Raster filters` controls how PnPInk handles this before PDF creation.

Mode	Behavior
`png`	Rasterizes filtered nodes to PNG before PDF export. This is the normal safe mode.
`jpeg`	Rasterizes filtered nodes to JPEG. This can reduce file size for photographic content.
`png_alfa`	Rasterizes filtered nodes to PNG with alpha support.
`inkscape`	Leaves filter handling to Inkscape's PDF exporter.
`none`	Does not pre-rasterize filters. Fastest, but filtered output can be wrong or missing.

The rasterization DPI is derived from the export DPI.

9.4 Other formats

Enable `Other formats` to export page-based output in a non-PDF format.

Supported formats:

`png`, `jpeg`, `jpeg2000`, `pdf`, `svg`, `tiff`, `webp`, `ps`, `eps`, `emf`, `wmf`

Use the `Pages` field to restrict export to specific pages. Syntax examples:

Pages value	Meaning
empty	Export all pages.
`1`	Export page 1.
`1,3-5,8`	Export pages 1, 3, 4, 5, and 8.

JPEG, TIFF, JPEG2000, and WebP can use a PNG intermediate plus Pillow conversion when that is the most reliable path for the current system.

9.5 SVG parts

Large generated SVG files can be split into smaller SVG parts. This is configured in the Preferences tab:

Control	Purpose
`Split SVG`	Enables chunked SVG output.
`Part target MB`	Target size for each SVG part.

Export detects existing chunked output and reuses it. This avoids rebuilding parts unnecessarily.

Smaller chunks can make Inkscape exports more stable and can increase parallelism, but very small chunks add overhead. For large decks, tune `Part target MB` by measuring the log instead of assuming that the smallest value is fastest.

9.6 Parallelism

The exporter can run multiple Inkscape shell workers in parallel. The preference is:

Key	Purpose
`inkscape_shell_workers`	Maximum number of parallel Inkscape shell workers used during export.

More workers can improve throughput on multi-core machines, but the best value depends on SVG complexity, disk speed, and available memory.

9.7 Automation profile

PnPInk launches Inkscape command-line export processes with an isolated automation environment. This keeps command-line exports from polluting the normal Inkscape `recently-used` data as much as possible.

The `Open SVG` GUI button is different from export: it opens the generated SVG for the user through the configured Inkscape launch path.

9.8 Preferences reference

The main export preferences are stored in `src/preferences.ini`.

Key	Values	Purpose
`export_pdf`	`0`, `1`	Enables standard PDF export.
`pdf_profiles`	comma-separated `default`, `screen`, `ebook`, `printer`, `prepress`	Standard PDF output profiles.
`export_pdfx`	`0`, `1`	Enables PDF/X CMYK export.
`pdfx_version`	`1a`, `3`, `4`	PDF/X standard used for CMYK export.
`pdf_cmyk_icc`	profile name or path	CMYK ICC profile.
`pdf_cmyk_pure_black_text`	`0`, `1`	Preserve text as pure black when supported.
`pdf_raster_mode`	`png`, `jpeg`, `png_alpha`, `inkscape`, `none`	Filter rasterization strategy.
`export_png`	`0`, `1`	Enables additional non-PDF export.
`export_other_format`	supported format name	Format used by `Other formats`.
`export_other_pages`	page list	Page filter for additional exports. Empty means all pages.
`export_dpi`	integer >= 1	Inkscape export DPI.
`export_jpeg_quality`	70-95	JPEG quality.
`split_svg_output`	`0`, `1`	Enables generated SVG parts.
`split_svg_chunk_mb`	integer >= 1	Target SVG part size in MB.
`inkscape_shell_workers`	integer >= 1	Parallel export workers.

9.9 Troubleshooting

If export cannot find the generated SVG, run `Generate` first and check that the output file exists.

If PDF/X export fails, verify that Ghostscript is installed and that the selected ICC profile exists.

If filtered images disappear or look wrong, use raster mode `png` or `png_alfa` before trying faster modes.

If export is slower after enabling SVG parts, increase `Part target MB`. Too many small chunks can cost more time than they save.

10 Extensions and Tools

This page summarizes the user-facing extensions present in the current project version.

10.1 Main generation tools

10.1.1 DeckMaker

Visible in Inkscape as:

`Extensions > PnPInk > DeckMaker`

Purpose:

- read CSV or Google Sheets data,
- expand snippets,
- clone templates,
- apply page/layout/source/fit rules,
- generate an editable SVG output,
- open the generated SVG in Inkscape,
- export PDF, PDF/X, and other output formats.

The GUI title shows the current DeckMaker version and the template file name. See DeckMaker GUI and Export.

10.1.2 Spritesheet

Visible in Inkscape as:

`Extensions > PnPInk > Spritesheet`

Purpose:

- open the interactive spritesheet GUI,
- define grid cuts and frame extraction workflows for atlas-like image sources.

10.2 Project/package tools

10.2.1 PnPInk ZVG Import / Export

File format: `.zvg`

Purpose:

- package an SVG project with local assets for reproducible sharing,
- reopen packaged work as a portable project.

10.2.2 PnPInk PNP Import / Export

File format: `.pnp`

Purpose:

- package a lighter regeneration-oriented project,
- preserve the SVG + dataset workflow with smaller payloads when possible.

See also Packages.

10.3 Utility tools

10.3.1 Preferences

Visible in Inkscape as:

`Extensions > PnPInk > Preferences`

Purpose:

- configure console and file log levels,
- configure JSON logging behavior,
- configure SVG output splitting,
- configure export and advanced generation options,
- persist preferences in `preferences.ini`.

10.3.2 Docs and Examples

Visible in Inkscape as:

`Extensions > PnPInk > Docs and Examples`

Purpose:

- open the installed PnPInk folder in the system browser,
- provide quick access to examples and local documentation.

10.4 Operational notes

10.4.1 Google Sheets authentication

The current codebase includes a PKCE-based Google Sheets client. The authentication flow is implemented in the project, but the operational setup is only lightly documented in the current docs set.

10.4.2 Logging

The project writes runtime logs to `src/pnpink.log`. This is useful when validating parser behavior, measuring generation/export time, or investigating mismatches between dataset and output.

10.4.3 Web-source caching

Remote web sources are cached locally and may later be included in ZVG packages depending on package mode and source type.

11 Snippets

Snippets are reusable text templates expanded before normal `${var}` replacement. This feature is designed to keep datasets readable when text patterns repeat.

11.1 What snippets are

Snippets are a compact mini-language inside PnPInk. They let you define short aliases that expand into longer fragments (text, SVG-rich text, paths, URL fragments, style fragments, and reusable token patterns).

Snippets are always processed first, immediately after DeckMaker reads dataset values and before normal variable substitution.

11.2 Syntax

This section defines the only supported definition format in the current engine.

Define snippets in comment lines:

```
# :Name(arg1 arg2=default) = template text
```

Rules from the current engine:

- No space is allowed between snippet name and `(`.
- Parameters are space-separated.
- Defaults use `name=value`.
- The replacement template is plain text after `=`.
- Old `->` definitions are not supported.

Call syntax:

```
:Name(value1 value2)
```

11.3 Arguments and Expansion

Argument mapping rules are what make snippets usable in real datasets.

Snippets support positional args, named args, defaults, and nesting.

```
# :Join(a b) = ${a}/${b}
# :Tf(text font size) = <tspan font-family='${font}'${size? font-size='${size}'}>${text}</tspan>

:Join(cards heroes)               -> cards/heroes
:Tf(Title Noto 16px)              -> <tspan font-family='Noto' font-size='16px'>Title</tspan>
:Tf(Title Noto)                   -> <tspan font-family='Noto'>Title</tspan>
:Tf(text=Title font=Noto size=12)  -> named arguments are valid
```

Conditional blocks use `${var? ...}` and render only when `var` is non-empty.

If a snippet has exactly one parameter, that parameter captures the full inner text:

```
# :Bold(text) = <b>${text}</b>
:Bold(This is bold text) -> <b>This is bold text</b>
```

11.4 Defaults and Optional Parts

Default values are used when a parameter is omitted.

```
# :Box(text color=black) = <rect stroke='${color}'/><text>${text}</text>

:Box(Title)     -> <rect stroke='black'/><text>Title</text>
:Box(Title red) -> <rect stroke='red'/><text>Title</text>
```

11.5 Quoting, Spaces, and Escaping

- Parameters containing spaces must use quotes.
- Use `\:` to keep a call literal.
- If a snippet does not exist, the call stays literal.
- If a call is malformed, it stays literal.

```
:Join("My Folder" file.svg) -> My Folder/file.svg
\:Join(a b)                 -> :Join(a b)
```

11.6 Nesting

Snippets can call other snippets and are expanded inside-out.

```
# :B(text) = <b>${text}</b>
# :C(text color) = <span fill='${color}'>${text}</span>

:C(:B(Name) red) -> <span fill='red'><b>Name</b></span>
```

11.7 Practical SVG Text Helpers

These helpers are optional, but they are the fastest way to build rich text consistently.

Useful helpers for `<tspan>` rich text:

```
# :Tb(text) = <tspan font-weight='bold'>${text}</tspan>
# :Ti(text) = <tspan font-style='italic'>${text}</tspan>
# :Ts(text) = <tspan text-decoration='underline'>${text}</tspan>
# :Tx(text) = <tspan text-decoration='line-through'>${text}</tspan>
# :Tc(text fill border bordercolor) = <tspan fill='${fill}'${border? stroke='${bordercolor}' stroke-width='${border}' paint-order='stroke'}>${text}</tspan>
# :Te(text) = <tspan font-family='Noto Color Emoji'>${text}</tspan>
```

Nested expansion is supported and resolved from inner to outer calls.

11.8 Processing Order

Processing order matters when snippets and `${var}` are combined in the same cell.

For each dataset cell:

1. Snippets are expanded.
2. `${var}` replacements are applied.
3. DSL placement and rendering run.

11.9 Variable Expressions

`${...}` placeholders are resolved after snippets. The expression may contain any text except nested braces, so the syntax remains open for future extensions.

Currently supported forms are intentionally small and predictable:

```
${name}
${name[0]}
${name[0].field}
```

This is mostly useful with internal variables created by sources, such as Wikimedia Commons category catalogs:

```
${_wkmcc1[0].title}
${_wkmcc1[0].url}
${_wkmcc1[0].thumburl}
```

Unsupported expressions resolve to an empty string instead of stopping the render.

11.10 Summary Table

Concept	Syntax	Example	Result
Definition	`# :Name(a b=def) = text`	`# :Hello(name) = Hi ${name}`	defines snippet
Use	`:Hello(World)`		`Hi World`
Default	`param=value`	`# :Box(t color=red)`	uses red if omitted
Conditional	`${p? text}`	`${size? font-size='${size}'}`	adds only if defined
Escape	`\Name()`		prevents expansion
Nested	`:A(:B(x))`		inner first
One parameter	`:Bold(any text)`		captures all
Quoted values	`:Join("My Folder" file)`		handles spaces

11.11 Design Principles

- Consistent: follows the same spacing and escaping rules as the rest of PnPInk.
- Readable: complex SVG text becomes short and maintainable.
- Composable: snippets can be nested and combined incrementally.
- Deterministic: plain text substitution rules, no runtime scripting.

12 Presets

This page is a lookup reference for standard page and shape identifiers accepted by the parser.

12.1 Page Presets (mm)

Preset	Size (mm)
A2	420 x 594
A3	297 x 420
A4	210 x 297
A5	148 x 210
A6	105 x 148
Letter	215.9 x 279.4
Legal	215.9 x 355.6
Tabloid	279.4 x 431.8

12.2 Shape Presets

Use these identifiers in `Layout{shape=...}` / `L{s=...}` and related shape-aware workflows.

Identifier	Aliases	Value (mm)
Standard	poker, magic, estandard, estandar	63 x 88
2.5x3.5inch	-	63.5 x 88.9
XL_Poker	xlpoker, xlpoker_, xlstandard, xlstandar, xl_standard, xl_standar	70.875 x 99.0
USA	bridge	56 x 87
Euro	mini	59 x 92
Asia	chimera	57.5 x 89
miniEuro	euromini	45 x 68
miniAsia	asiamini, minichimera, chimeramini	43 x 65
miniUSA	usamini	41 x 63
Tarot	-	70 x 120
FrenchTarot	-	61 x 112
Volcano	-	70 x 110
Wonder	-	65 x 100
Spanish	baraja	61 x 95
Desert	-	50 x 65
squareS	-	50 x 50
square	-	70 x 70
squareL	-	100 x 100
Counter38	counter38, inch38	9.525 x 9.525
Counter12	counter12, inch12	12.7 x 12.7
Counter34	counter34, inch34	19.05 x 19.05
Counter1	counter1, inch1	25.4 x 25.4
Dixit	-	80 x 120
CreditCard	creditcard, cr80, id1, id-1	85.6 x 54

Also accepted (normalization rules):

- Matching is case-insensitive.
- Accents are normalized.
- Spaces, hyphens, underscores, and dots are ignored.

Examples:

- `STANDARD`, `standard`, `StAnDaRd` -> `Standard`
- `mini usa`, `mini-usa`, `mini_usa` -> `miniUSA`
- `id1`, `ID-1`, `id_1`, `id.1` -> `ID-1`
- `xl-poker`, `XL_POKER`, `xl poker` -> `XL_Poker`
- `counter_1/2`, `counter_1_2`, `COUNTER12` -> `Counter12`
- `inch_3/8`, `inch3_8`, `INCH38` -> `Counter38`