Schemdraw

Release 0.23

Collin J. Delker

May 29, 2026

CONTENTS:

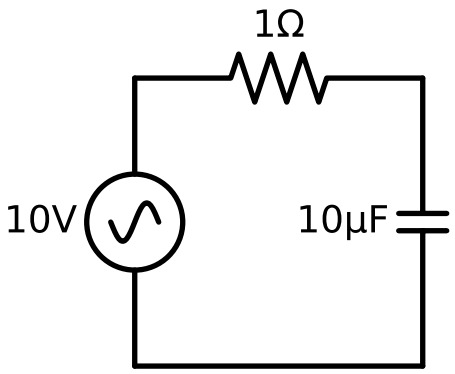
Schemdraw is a Python package for producing high-quality electrical circuit schematic diagrams. Circuit elements are added, one at a time, similar to how you might draw them by hand, using Python methods.

For example,

```
with schemdraw.Drawing():
    elm.Resistor().right().label('1Ω')
```

creates a new schemdraw drawing with a resistor going to the right with a label of "1Ω". The next element added to the drawing will start at the endpoint of the resistor.

```
with schemdraw.Drawing():
    elm.Resistor().right().label('1Ω')
    elm.Capacitor().down().label('10μF')
    elm.Line().left()
    elm.SourceSin().up().label('10V')
```



CHAPTER

ONE

GETTING STARTED

1.1 Installation

schemdraw can be installed from pip using

```
pip install schemdraw
```

or to include optional `matplotlib` backend dependencies:

```
pip install schemdraw[matplotlib]
```

To allow the SVG drawing *Backends* to render math expressions, install the optional ziamath dependency with:

```
pip install schemdraw[svgmath]
```

Alternatively, schemdraw can be installed directly by downloading the source and running

```
pip install ./
```

Schemdraw requires Python 3.9 or higher.

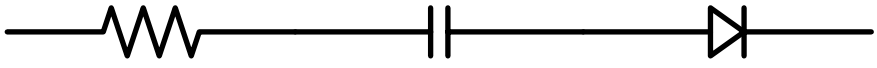
1.2 Overview

The `schemdraw` module allows for drawing circuit elements. `schemdraw.elements` contains *Basic Elements* predefined for use in a drawing. A common import structure is:

```
import schemdraw
import schemdraw.elements as elm
```

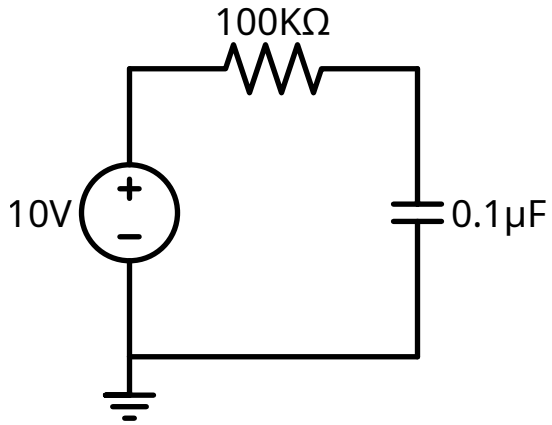
To make a circuit diagram, use a context manager (*with* statement) on a *schemdraw.Drawing*. Then any *schemdraw.elements.Element* instances created within the *with* block added to the drawing:

```
with schemdraw.Drawing():
    elm.Resistor()
    elm.Capacitor()
    elm.Diode()
```



Element placement and other properties are set using a chained method interface, for example:

```
with schemdraw.Drawing():
    elm.Resistor().label('100KΩ')
    elm.Capacitor().down().label('0.1μF', loc='bottom')
    elm.Line().left()
    elm.Ground()
    elm.SourceV().up().label('10V')
```



Methods *up*, *down*, *left*, *right* specify the drawing direction, and *label* adds text to the element. If not specified, elements reuse the same direction from the previous element, and begin where the previous element ended.

Using the *with* context manager is a convenience, letting the drawing be displayed and saved upon exiting the *with* block. Schematics may also be created by assinging the new Drawing instance to a variable and adding elements to the drawing with *d.add* or *d* +=, then calling *draw()* and/or *save()* to show the drawing:

```
d = schemdraw.Drawing()
d += elm.Resistor()
...
d.draw()
d.save('my_circuit.svg')
```

For full details of placing and stylizing elements, see *Placing Elements*. and `schemdraw.elements.Element`.

In general, parameters that control **what** is drawn are passed to the element itself, and parameters that control **how** things are drawn are set using chained Element methods. For example, to make a polarized Capacitor, pass *polar=True* as an argument to *Capacitor*, but to change the Capacitor's color, use the *.color()* method: *elm.Capacitor(polar=True).color('red')*.

1.3 Viewing the Drawing

1.3.1 Jupyter

When run in a Jupyter notebook, the schematic will be drawn to the cell output after the *with* block is exited. If your schematics pop up in an external window, and you are using the Matplotlib backend, set Matplotlib to inline mode before importing schemdraw:

```
%matplotlib inline
```

For best results when viewing circuits in the notebook, use a vector figure format, such as svg before importing schemdraw:

```
%config InlineBackend.figure_format = 'svg'
```

1.3.2 Python Scripts and GUI/Web apps

If run as a Python script, the schematic will be opened in a pop-up window after the *with* block exits. Add the *show=False* option when creating the Drawing to suppress the window from appearing.

```
with schemdraw.Drawing(show=False) as d:
    ...
```

The raw image data as a bytes array can be obtained by calling *.get_imagedata()* with the after the *with* block exits. This can be useful for integrating schemdraw into an existing GUI or web application.

```
with schemdraw.Drawing() as drawing:
    ...
image_bytes = drawing.get_imagedata('svg')
```

1.3.3 Headless Servers

When running on a server, sometimes there is no display available. The code may attempt to open the GUI preview window and fail. In these cases, try setting the Matplotlib backend to a non-GUI option. Before importing schemdraw, add these lines to use the Agg backend which does not have a GUI. Then get the drawing using *d.get_imagedata()*, or *d.save()* to get the image.

```
import matplotlib
matplotlib.use('Agg') # Set Matplotlib's backend here
```

Alternatively, use Schemdraw's SVG backend (see *Backends*).

1.4 Saving Drawings

To save the schematic to a file, add the *file* parameter when setting up the Drawing. The image type is determined from the file extension. Options include *svg*, *eps*, *png*, *pdf*, and *jpg* when using the Matplotlib backend, and *svg* when using the SVG backend. A vector format such as *svg* is recommended for best image quality.

```
with schemdraw.Drawing(file='my_circuit.svg') as d:
    ...
```

The Drawing may also be saved using with the *schemdraw.Drawing.save()* method.

CHAPTER

TWO

USAGE

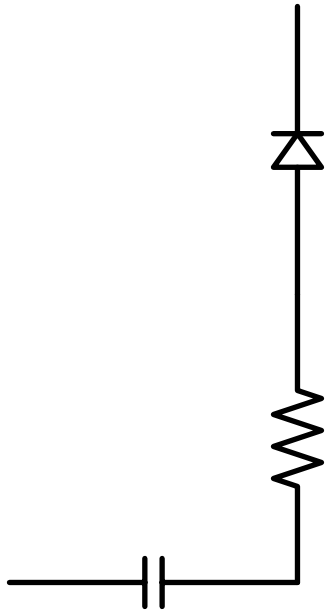
2.1 Placing Elements

Elements instantiated insde a *with* block are automatically added to the Drawing. The Drawing maintains a current position and direction, such that the default placement of the next element will start at the end of the previous element, going in the same direction.

```
with schemdraw.Drawing():
    elm.Capacitor()
    elm.Resistor()
    elm.Diode()
```

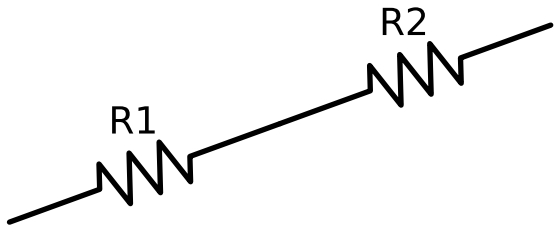
If a direction method (*up*, *down*, *left*, *right*) is added to an element, the element is rotated in that direction, and future elements take the same direction:

```
with schemdraw.Drawing():
    elm.Capacitor()
    elm.Resistor().up()
    elm.Diode()
```



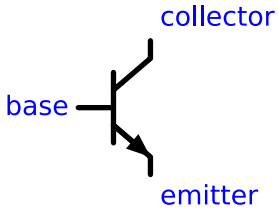
The *theta* method can be used to specify any rotation angle in degrees.

```
with schemdraw.Drawing():
    elm.Resistor().theta(20).label('R1')
    elm.Resistor().label('R2')  # Takes position and direction from R1
```



2.1.1 Anchors and Positioning

All elements have a set of predefined anchor positions within the element. For example, a bipolar transistor has *base*, *emitter*, and *collector* anchors. All two-terminal elements have anchors named *start*, *center*, and *end*. Anchor names are shown for each element in *Circuit Elements*. They are also listed in the docstring for each element.



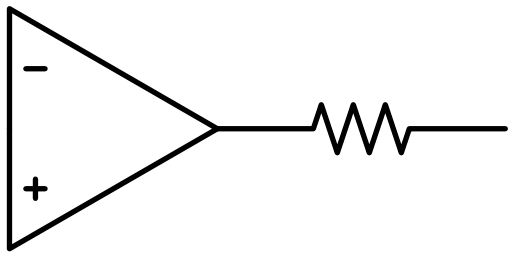
Once an element is added to the drawing, all its anchor positions will be added as attributes to the element object, so the base position of transistor assigned to variable *Q* may be accessed via *Q.base*. Alternatively, anchor positions may be accessed using brackets: *Q['base']*. This option is useful when anchors are generated that are not valid Python identifiers, for example when the anchor name may contain a dot or other symbol. The returned position is relative to the drawing origin, and will only be accessible once an element is placed in a drawing.

The element's *anchors* attribute contains a dictionary of all anchors relative to the element's origin and can be accessed before placement.

At Method

To place an element at a specific location connected to another element, use the *at* method. For example, to draw an opamp and place a resistor on its output, store the Opamp instance to a variable. Then call the *at* method of the new element passing the *Opamp.out* anchor. After the resistor is drawn, the current drawing position is moved to the endpoint of the resistor.

```
with schemdraw.Drawing():
    opamp = elm.Opamp()
    elm.Resistor().right().at(opamp.out)
```

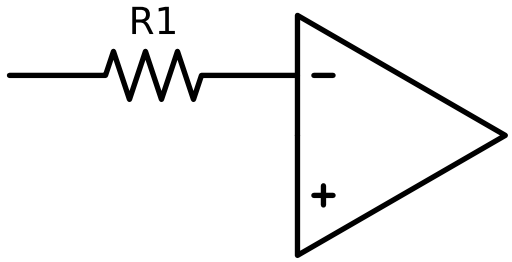


Alignment

Anchors are also used to align new elements with respect to existing elements.

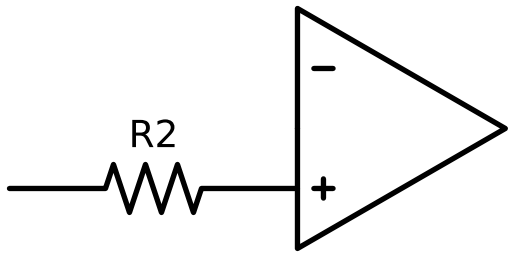
Suppose a resistor has just been placed, and now an Opamp should be connected to the resistor. The *schemdraw.elements.Element.anchor()* method tells the Drawing which anchor on the Opamp should align with resistor. Here, an Opamp is placed at the end of a resistor, connected to the opamp's *in1* anchor (the inverting input).

```
with schemdraw.Drawing():
    elm.Resistor().label('R1')
    elm.Opamp().anchor('in1')  # Place the `in1` anchor at the current drawing position
```



Compared to anchoring the opamp at *in2* (the noninverting input):

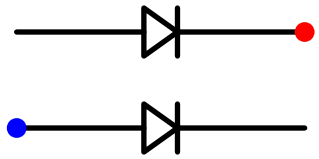
```
with schemdraw.Drawing():
    elm.Resistor().label('R2')
    elm.Opamp().anchor('in2')  # Place the `in2` anchor at the current drawing position
```



Hold method

To place an element without moving the drawing position, use the element's *schemdraw.elements.Element.hold()* method. The element will be placed without changing the drawing state.

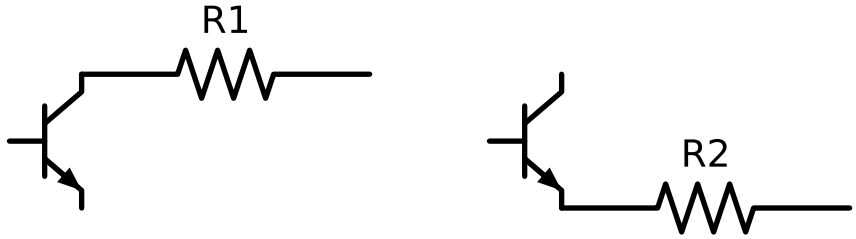
```
with schemdraw.Drawing() as d:
    elm.Diode()  # Normal placement: drawing position moves to end of element
    elm.Dot().color('red')

    d.move(dx=-d.unit, dy=-1)
    elm.Diode().hold()  # Hold method prevents position from changing
    elm.Dot().color('blue')
```



Drop Method

Some elements, especially those with more than 2 terminals, do not necessarily leave the drawing position where desired, so after drawing an element, the current drawing position can be set using the *schemdraw.elements.Element.drop()* method to specify the anchor at which to leave the drawing cursor.

```
with schemdraw.Drawing() as d:
    bjt1 = elm.BjtNpn()
    elm.Resistor().label('R1')  # Default cursor placement after placing BJT

    d.move_from(bjt1.base, dx=5)
    bjt2 = elm.BjtNpn().drop('emitter')  # Leave the cursor on the emitter after placing␣
↪BJT
    elm.Resistor().label('R2')
```

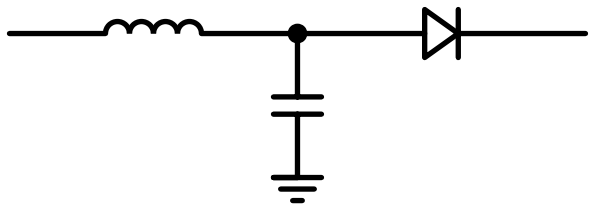


Drawing State

The *schemdraw.Drawing* maintains a drawing state that includes the current x, y position, stored in the *Drawing.here* attribute as a (x, y) tuple, and drawing direction stored in the *Drawing.theta* attribute. The *schemdraw.Drawing.hold()* method returns a context manager that saves the drawing state to a LIFO stack, with the state restored when the context manager exits.

```
with schemdraw.Drawing() as d:
    elm.Inductor()
    elm.Dot()
    print('d.here:', d.here)
    with d.hold():  # Save this drawing position/direction for later
        elm.Capacitor().down(1.5)  # Go off in another direction temporarily
        elm.Ground(lead=False)
        print('d.here:', d.here)

    print('d.here:', d.here)   # Drawing state restored when the with-block exits
    elm.Diode()
```

```
d.here: Point(3.0,0.0)
d.here: Point(2.9999999999999996,-1.5)
d.here: Point(3.0,0.0)
```

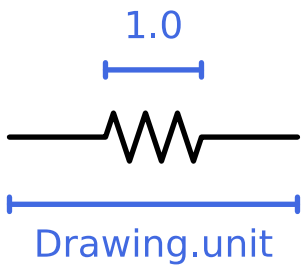


Alternatively, the *schemdraw.Drawing.push()* and *schemdraw.Drawing.pop()* methods work in the same way without the context manager.

Changing the drawing position can be accomplished by calling *schemdraw.Drawing.move()* or *schemdraw.Drawing.move_from()*.

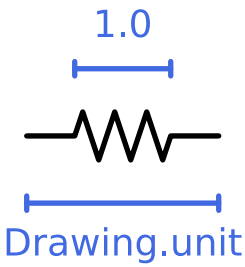
2.1.2 Dimensions

The inner zig-zag portion of a resistor has length of 1 unit, while the default lead extensions are 1 unit on each side, making the default total resistor length 3 units. Placement methods such as *at* and *to* accept a tuple of (x, y) position in these units.



This default 2-terminal length can be changed using the *unit* parameter to the *schemdraw.Drawing.config()* method:

```
with schemdraw.Drawing() as d:
    d.config(unit=2)
    ...
```

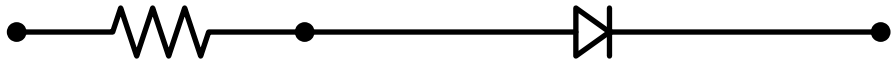


2.1.3 Two-Terminal Elements

In Schemdraw, a "Two-Terminal Element" is any element that can grow to fill a given length (this includes elements such as the Potentiometer, even though it electrically has three terminals). All two-terminal elements subclass *schemdraw.elements.Element2Term*. They have some additional methods for setting placement and length.

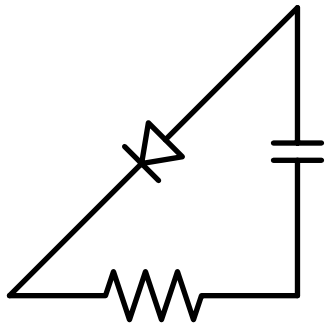
The *length* method sets an exact length for a two-terminal element. Alternatively, the *up*, *down*, *left*, and *right* methods on two-terminal elements take a length parameter.

```
with schemdraw.Drawing() as d:
    elm.Dot()
    elm.Resistor()
    elm.Dot()
    elm.Diode().length(6)
    elm.Dot()
```



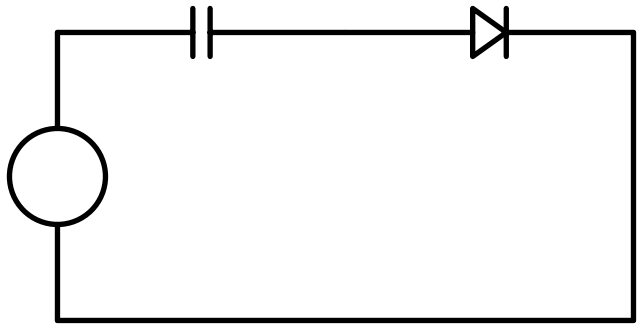
The *to* method will set an exact endpoint for a 2-terminal element. The starting point is still the ending location of the previous element. Notice the Diode is stretched longer than the standard element length in order to fill the diagonal distance.

```
with schemdraw.Drawing() as d:
    R = elm.Resistor()
    C = elm.Capacitor().up()
    Q = elm.Diode().to(R.start)
```



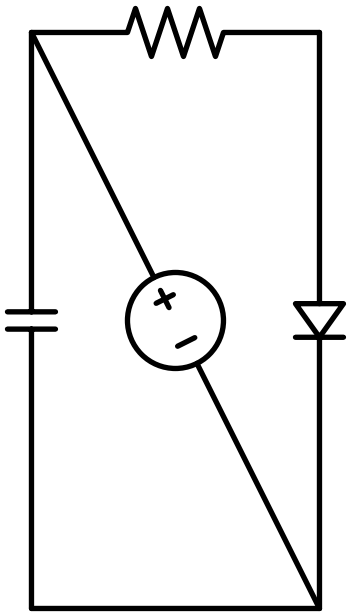
The *tox* and *toy* methods are useful for placing 2-terminal elements to "close the loop", without requiring an exact length. They extend the element horizontally or vertically to the x- or y- coordinate of the anchor given as the argument. These methods automatically change the drawing direction. Here, the Line element does not need to specify an exact length to fill the space and connect back with the Source.

```
with schemdraw.Drawing():
    C = elm.Capacitor()
    elm.Diode()
    elm.Line().down()

    # Now we want to close the loop, but can use `tox`
    # to avoid having to know exactly how far to go.
    # The Line will extend horizontally to the same x-position
    # as the Capacitor's `start` anchor.
    elm.Line().tox(C.start)

    # Now close the loop by relying on the fact that all
    # two-terminal elements (including Source and Line)
    # are the same length by default
    elm.Source().up()
```



Finally, exact endpoints can also be specified using the *endpoints* method.

```
with schemdraw.Drawing():
    R = elm.Resistor()
    Q = elm.Diode().down(6)
    elm.Line().tox(R.start)
    elm.Capacitor().toy(R.start)
    elm.SourceV().endpoints(Q.end, R.start)
```



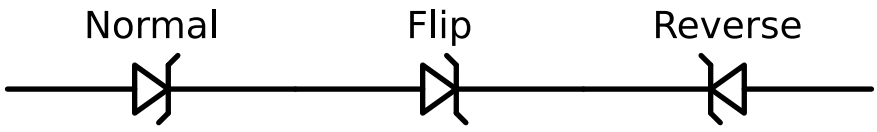
Two-terminal elements are centered between their endpoints. Occasionally, it is useful to place them off-center. Use the *shift* method with a value between -1 and 1 to shift the element to one side.

```
with schemdraw.Drawing():
    elm.Resistor().shift(-0.4).dot()
    elm.Resistor().shift(0.75)
```

2.1.4 Orientation

The *flip* and *reverse* methods change orientation of directional elements such as Diodes, but they do not affect the drawing direction.

```
with schemdraw.Drawing():
    elm.Zener().label('Normal')
    elm.Zener().flip().label('Flip')
    elm.Zener().reverse().label('Reverse')
```



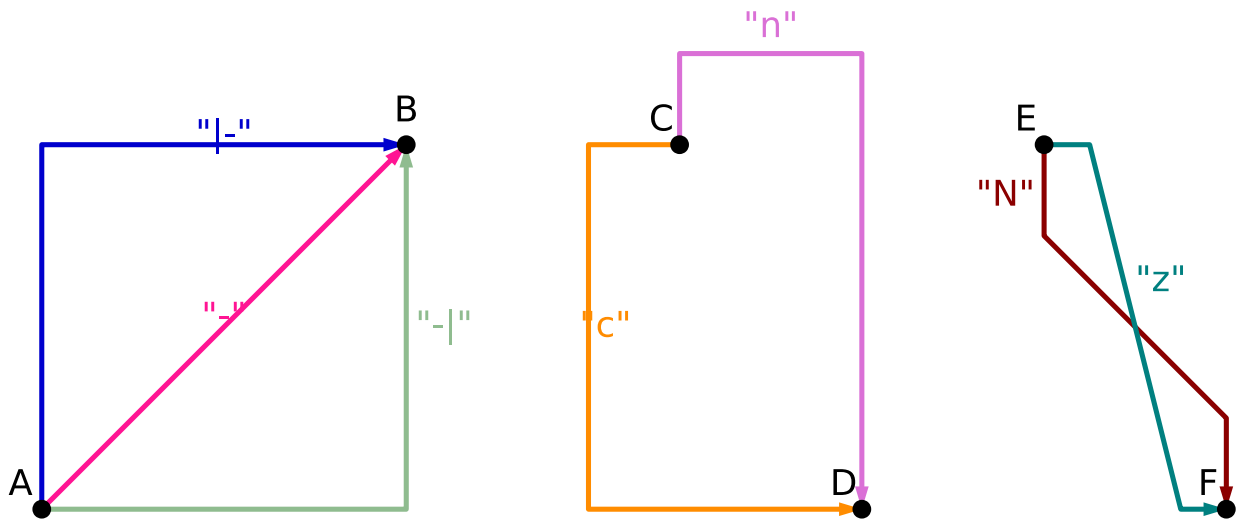
2.1.5 Connecting Elements

Typically, the *schemdraw.elements.lines.Line* element is used to connect elements together. More complex line routing requires multiple Line elements. The *schemdraw.elements.lines.Wire* element is used as a shortcut for placing multiple connecting lines at once. The Wire element connects the start and end points based on its *shape* parameter. The *k* parameter is used to set the distance before the wire first changes direction.

Table 1: Wire Shape Parameters

Shape Parameter	Description
`-`	Direct Line
`-|`	Horizontal then vertical
`|-`	Vertical then horizontal
`|-|` or `n`	Vertical-horizontal-vertical (like an n or u)
`-|-` or `c`	Horizontal-vertical-horizontal (like a c or ⊃)
`-/-` or `z`	Horizontal-diagonal-horizontal
`|/|` or `N`	Vertical-diagonal-vertical

```
elm.Wire('-', arrow='->').at(A.center).to(B.center).color('deeppink').label('"-"')
elm.Wire('|-', arrow='->').at(A.center).to(B.center).color('mediumblue').label('"|-"')
elm.Wire('-|', arrow='->').at(A.center).to(B.center).color('darkseagreen').label('"-|"')
elm.Wire('c', k=-1, arrow='->').at(C.center).to(D.center).color('darkorange').label('"c"
↪', halign='left')
elm.Wire('n', arrow='->').at(C.center).to(D.center).color('orchid').label('"n"')
elm.Wire('N', arrow='->').at(E.center).to(F.center).color('darkred').label('"N"', 'start
↪', ofst=(-.1, -.75))
elm.Wire('z', k=.5, arrow='->').at(E.center).to(F.center).color('teal').label('"z"',␣
↪halign='left', ofst=(0, .5))
```



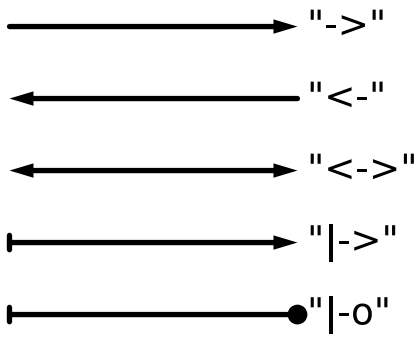
Both *Line* and *Wire* elements take an *arrow* parameter, a string specification of arrowhead types at the start and end of the wire. The arrow string may contain "<", ">", for arrowheads, "|" for an endcap, and "o" for a dot. Some examples are shown below:

```
with schemdraw.Drawing():
    elm.Line(arrow='->').label('"->"', 'right')
```

(continues on next page)

(continued from previous page)

```
    elm.Line(arrow='<-').at((0, -.75)).label('"<-"', 'right')
    elm.Line(arrow='<->').at((0, -1.5)).label('"<->"', 'right')
    elm.Line(arrow='|->').at((0, -2.25)).label('"|->"', 'right')
    elm.Line(arrow='|-o').at((0, -3.0)).label('"|-o"', 'right')
```



Because dots are used to show connected wires, all two-terminal elements have *dot* and *idot* methods for quickly adding a dot at the end or beginning of the element, respectively.

```
elm.Resistor().dot()
```

2.1.6 Keyword Arguments

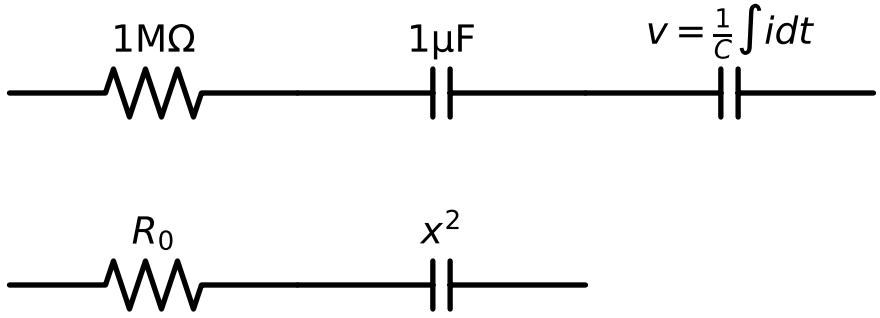
All *schemdraw.elements.Element* types take keyword arguments that can also be used to set element properties, partly for historical reasons but also for easy element setup via dictionary unpacking. The keyword arguments are equivalent to calling the Element setup methods. The keyword arguments are not validated or type checked, so the chained method interface described above is recommended for configuring elements.

Keyword Argument	Method Equivalent
d='up'	*.up()*
d='down'	*.down()*
d='left'	*.left()*
d='right'	*.right()*
theta=X	*.theta(X)*
at=X or *xy=X*	*.at(X)*
flip=True	*.flip()*
reverse=True	*.reverse()*
anchor=X	*.anchor(X)*
zoom=X	*.scale(X)*
color=X	*.color(X)*
fill=X	*.fill(X)*
ls=X	*.linestyle(X)*
lw=X	*.linewidth(X)*
zorder=X	*.zorder(X)*
move_cur=False	*.hold()*
label=X	*.label(X)*
botlabel=X	*.label(X, loc='bottom')*
lftlabel=X	*.label(X, loc='left')*
rgtlabel=X	*.label(X, loc='right')*
toplabel=X	*.label(X, loc='top')*
lblloc=X	*.label(..., loc=X)*

2.2 Labels

Labels are added to elements using the *schemdraw.elements.Element.label()* method. Some unicode utf-8 characters are allowed, such as `'1μF'` and `'1MΩ'` if the character is included in your font set. Alternatively, full LaTeX math expressions can be rendered when enclosed in *$..$* For a description of supported math expressions, in the Matplotlib backend see Matplotlib Mathtext, and the SVG backend refer to the Ziamath package. Subscripts and superscripts are also added using LaTeX math mode, enclosed in *$..$*:

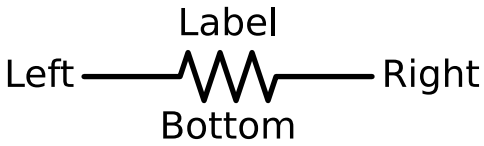
```
with schemdraw.Drawing():
    elm.Resistor().label('1MΩ')
    elm.Capacitor().label('1μF')
    elm.Capacitor().label(r'$v = \frac{1}{C} \int i dt$')
    elm.Resistor().at((0, -2)).label('$R_0$')
    elm.Capacitor().label('$x^2$')
```



2.2.1 Location

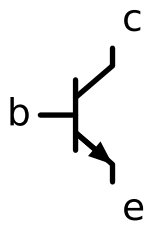
The label location is specified with the *loc* parameter to the *label* method. It can be *left*, *right*, *top*, *bottom*, or the name of a defined anchor within the element. These directions do not depend on rotation. A label with *loc='left'* is always on the leftmost terminal of the element.

```
with schemdraw.Drawing():
    (elm.Resistor()
        .label('Label')  # 'top' is default
        .label('Bottom', loc='bottom')
        .label('Right', loc='right')
        .label('Left', loc='left'))
```



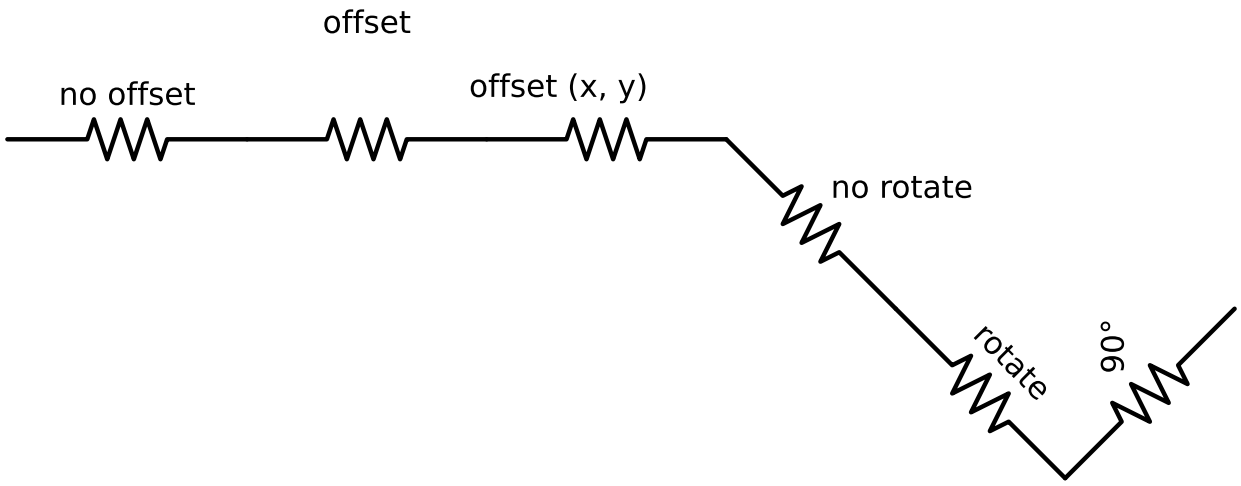
Labels may also be placed near an element anchor by giving the anchor name as the *loc* parameter.

```
with schemdraw.Drawing():
    (elm.BjtNpn()
        .label('b', loc='base')
        .label('c', loc='collector')
        .label('e', loc='emitter'))
```



The *schemdraw.elements.Element.label()* method also takes parameters that control the label's rotation, offset, font, alignment, and color. Label text stays horizontal by default, but may be rotated to the same angle as the element using *rotate=True*, or any angle *X* in degrees with *rotate=X*. Offsets apply vertically if a float value is given, or in both x and y if a tuple is given.

```
with schemdraw.Drawing():
    elm.Resistor().label('no offset')
    elm.Resistor().label('offset', ofst=1)
    elm.Resistor().label('offset (x, y)', ofst=(-.6, .2))
    elm.Resistor().theta(-45).label('no rotate')
    elm.Resistor().theta(-45).label('rotate', rotate=True)
    elm.Resistor().theta(45).label('90°', rotate=90)
```



Labels may also be added anywhere using the *schemdraw.elements.lines.Label* element. The element itself draws nothing, but labels can be added to it:

```
elm.Label().label('Hello')
```

2.2.2 Voltage Labels

A label may also be a list/tuple of strings, which will be evenly-spaced along the length of the element. This allows for labeling positive and negative along with a component name, for example:

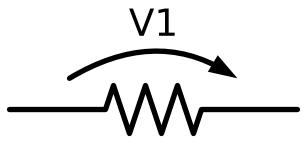
```
elm.Resistor().label(('-','$V_1$','+'))  # Note: using endash U+2013 character
```

Use the *Gap* element to label voltage across a terminal:

```
with schemdraw.Drawing():
    elm.Line().dot(open=True)
    elm.Gap().label(('-','$V_o$','+'))
    elm.Line().idot(open=True)
```

A voltage label drawn as an arc over the element may be added using *schemdraw.elements.lines.VoltageLabelArc*.

```
with schemdraw.Drawing():
    R1 = elm.Resistor()
    elm.VoltageLabelArc().at(R1).label('V1')
```

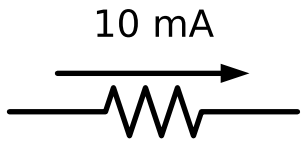


2.2.3 Current Arrow Labels

Current Arrow

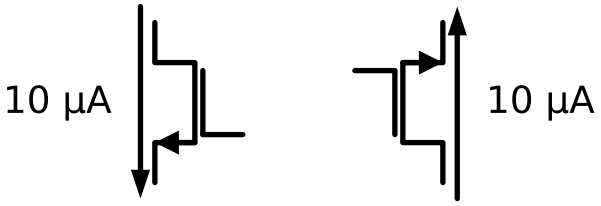
To label the current through an element, the *schemdraw.elements.lines.CurrentLabel* element can be added. The *at* method of this element can take an Element instance to label, and the arrow will be placed over the center of that Element.

```
with schemdraw.Drawing():
    R1 = elm.Resistor()
    elm.CurrentLabel().at(R1).label('10 mA')
```



For transistors, the label will follow sensible bias currents by default.

```
with schemdraw.Drawing() as d:
    Q1 = elm.AnalogNFet()
    elm.CurrentLabel().at(Q1).label('10 µA')
    d.move(3)
    Q2 = elm.AnalogNFet().flip().reverse()
    elm.CurrentLabel().at(Q2).label('10 µA')
```



Inline Current Arrow

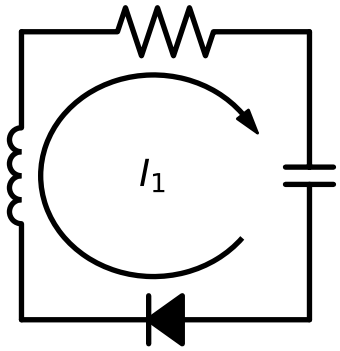
Alternatively, current labels can be drawn inline as arrowheads on the leads of 2-terminal elements using *schemdraw.elements.lines.CurrentLabelInline*. Parameters *direction* and *start* control whether the arrow is shown pointing into or out of the element, and which end to place the arrowhead on.

```
with schemdraw.Drawing():
    R1 = elm.Resistor()
    elm.CurrentLabelInline(direction='in').at(R1).label('10 mA')
```

Loop Current

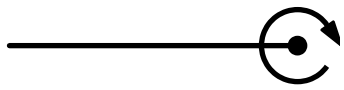
Loop currents can be added using *schemdraw.elements.lines.LoopCurrent*, given a list of 4 existing elements surrounding the loop.

```
with schemdraw.Drawing():
    R1 = elm.Resistor()
    C1 = elm.Capacitor().down()
    D1 = elm.Diode().fill(True).left()
    L1 = elm.Inductor().up()
    elm.LoopCurrent([R1, C1, D1, L1], direction='cw').label('$I_1$')
```



Alternatively, loop current arrows can be added anywhere with any size using *schemdraw.elements.lines.LoopArrow*.

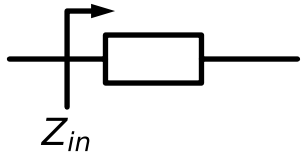
```
with schemdraw.Drawing():
    a = elm.Line().dot()
    elm.LoopArrow(width=.75, height=.75).at(a.end)
```



Impedance Arrow Label

A right-angle arrow label, often used to indicate impedance looking into a node, is added using *schemdraw.elements.lines.ZLabel*.

```
with schemdraw.Drawing():
    R = elm.RBox().right()
    elm.ZLabel().at(R).label('$Z_{in}$')
```

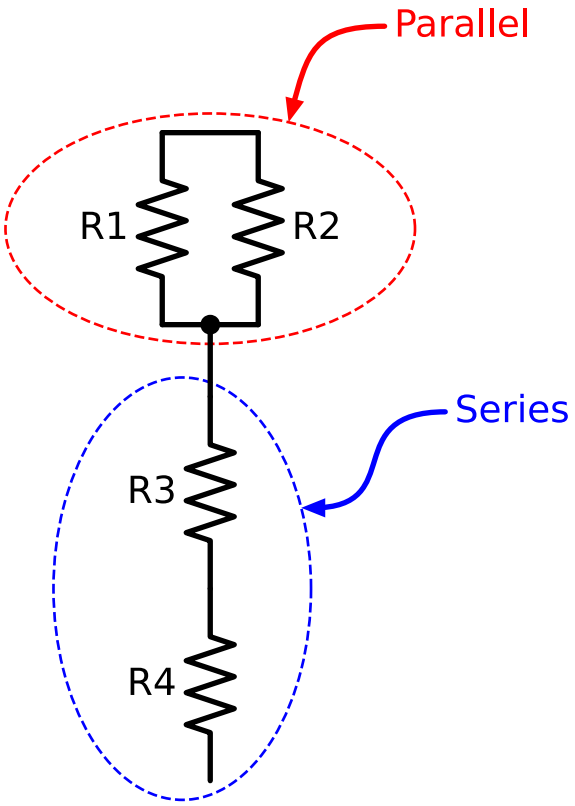


2.2.4 Annotations

To make text and arrow annotations to a schematic, the *schemdraw.elements.lines.Annotate* element draws a curvy arrow with label placed at it's end. It is based on the *schemdraw.elements.lines.Arc3* element.

The *schemdraw.elements.lines.Encircle* and *schemdraw.elements.lines.EncircleBox* elements draw an ellipse, or rounded rectangle, surrounding a list of elements.

```
parallel = elm.Encircle([R1, R2], padx=.8).linestyle('--').linewidth(1).color('red')
series = elm.Encircle([R3, R4], padx=.8).linestyle('--').linewidth(1).color('blue')

elm.Annotate().at(parallel.NNE).delta(dx=1, dy=1).label('Parallel').color('red')
elm.Annotate(th1=0).at(series.ENE).delta(dx=1.5, dy=1).label('Series').color('blue')
```



2.2.5 Links

In the SVG backend and *text* mode, hyperlinks may be added to labels using the *href* attribute. This example adds a hyperlink to the top of this page.

```
schemdraw.use('svg')
schemdraw.svgconfig.text = 'text'

with schemdraw.Drawing() as d:
    elm.Resistor().label(r'Link to Top', href="#top", color="blue")
```

Due to browser restrictions, SVG images having links used in HTML files must be included inline as <*svg*> tags in the html, not embedded as images with <*img src=”figure.svg”*>.

2.2.6 Decoration

Text decoration (underline, overline, and strike-through) may also be added in the SVG backend:

```
schemdraw.use('svg')
schemdraw.svgconfig.text = 'text'

with schemdraw.Drawing():
    elm.Resistor().label('Underline', decoration='underline')
    elm.Resistor().label('Overline', decoration='overline')
    elm.Resistor().label('Strike Through', decoration='line-through')
```

2.3 Styling

Style options, such as color, line thickness, and fonts, may be set at the global level (all Schemdraw Drawings), at the Drawing level, or on individual Elements.

2.3.1 Individual Elements

Element styling methods include *color*, *fill*, *linewidth*, and *linestyle*. If a style method is not called when creating an Element, its value is obtained from from the drawing or global defaults.

Color and fill parameters accept any named SVG color or a hex color string such as '#6A5ACD'. Linestyle parameters may be '-', '–', ':', or '-.'.

```
# All elements are blue with lightgray fill unless specified otherwise
with schemdraw.Drawing(color='blue', fill='lightgray'):
    elm.Diode()
    elm.Diode().fill('red')        # Fill overrides drawing color here
    elm.Resistor().fill('purple')  # Fill has no effect on non-closed elements
    elm.RBox().linestyle('--').color('orange')
    elm.Resistor().linewidth(5)
```

The *label* method also accepts color, font, and fontsize parameters, allowing labels with different style as their elements.

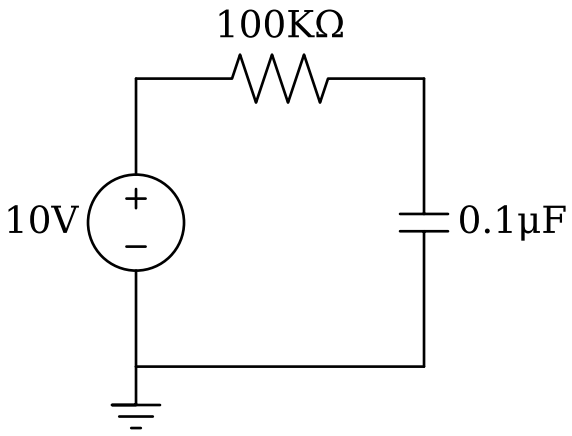
2.3.2 Drawing style

Styles may be applied to an entire drawing using the *schemdraw.Drawing.config()* method. These parameters include color, linewidth, font, fontsize, linestyle, fill, and background color. Additionally, the *config* method allows specification of the default 2-Terminal element length.

2.3.3 Global style

Styles may be applied to every new drawing created by Schemdraw (during the Python session) using *schemdraw.config()*, using the same arguments as the Drawing config method.

```
schemdraw.config(lw=1, font='serif')
with schemdraw.Drawing():
    elm.Resistor().label('100KΩ')
    elm.Capacitor().down().label('0.1μF', loc='bottom')
    elm.Line().left()
    elm.Ground()
    elm.SourceV().up().label('10V')
```

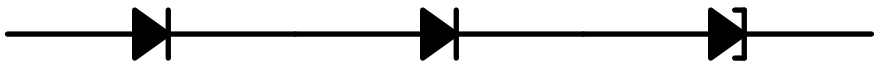


2.3.4 Global Element Configuration

Each Element has a *defaults* dictionary attribute containing default parameters for drawing the element.

For example, to fill all Diode elements:

```
elm.Diode.defaults['fill'] = True
with schemdraw.Drawing():
    elm.Diode()
    elm.Diode()
    elm.DiodeTunnel()
```



Notice that the defaults apply to Diode and all elements subclassed from Diode, such as DiodeTunnel. In general, the docstring of each Element lists arguments with their default values, these arguments may be specified in the *defaults* dictionary.

Styling Hierarchy

Element styles are applied in order of preference:

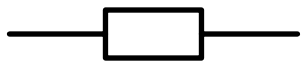
1) Setter methods like *.fill()* or *.color()* called after the Element is instantiated
2) Keyword arguments provided to Element instantiation
3) Defaults set by user in Element.defaults (inheriting from parent classes)
4) Parameters overridden by the Element definition
5) Parameters set in Drawing.config
6) Parameters set by Schemdraw.config

2.3.5 U.S. versus European Style

The *schemdraw.elements.Element.style()* method will to reconfigure elements in IEEE/U.S. style or IEC/European style. The *schemdraw.elements.STYLE_IEC* and *schemdraw.elements.STYLE_IEEE* are dictionaries for use in the *style* method to change configuration of various elements that use different standard symbols (resistor, variable resistor, photo resistor, etc.)

To configure IEC/European style, use the *style* method with the *elm.STYLE_IEC* dictionary.

```
elm.style(elm.STYLE_IEC)
elm.Resistor()
```



```
elm.style(elm.STYLE_IEEE)
elm.Resistor()
```

To see all the elements that change between IEEE and IEC, see *Resistors*.

2.3.6 Fonts

The font for label text may be set using the *font* parameter, either in the *schemdraw.elements.Element.label()* method for a single label, or in *schemdraw.Drawing.config()* to set the font for the entire drawing. The font parameter may be a string containing the name of a font installed in the system fonts path, a path to a TTF or OTF font file, or the name of a font family such as "serif" or "sans". These font options apply whether working in the Matplotlib or SVG backends.

```
with schemdraw.Drawing():
    # Default font
    elm.RBox().label('R1\n500K')

    # Named font in system fonts path
    elm.RBox().label('R1\n500K', font='Comic Sans MS')

    # Path to a TTF file
```

(continues on next page)

(continued from previous page)

```
    elm.RBox().label('R1\n500K', font='Peralta-Regular.ttf')

    # Font family
    elm.RBox().label('R1\n500K', font='serif')
```

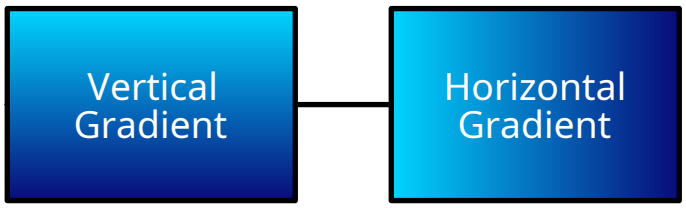
For typesetting math expressions, the *mathfont* parameter is used. In the Matplotlib backend, a limited selection of math fonts are available. With the SVG backend in the *path* text mode, the mathfont parameter may be the path to any TTF or OTF file that contains a MATH table (requires Ziamath).

```
with schemdraw.Drawing(canvas='svg'):
    # Default math font
    elm.RBox().label(r'$\sqrt{a^2+b^2}$').at((0, -2))

    # Path to a TTF file with MATH font table (SVG backend only)
    elm.RBox().label(r'$\sqrt{a^2+b^2}$', mathfont='Asana-Math.ttf')
```

2.3.7 Gradient Fill

Elements may be filled with a gradient using *gradient_fill*, providing the two colors and a boolean specifying whether to fade vertically or horizontally. Gradients are only supported in the SVG backend.

```
from schemdraw import flow
with schemdraw.Drawing():
    flow.Box().gradient_fill('#00D4FF', '#090979', True).label('Vertical\nGradient',␣
↪color='white')
    flow.Line().length(1)
    flow.Box().gradient_fill('#00D4FF', '#090979', False).label('Horizontal\nGradient',␣
↪color='white')
```



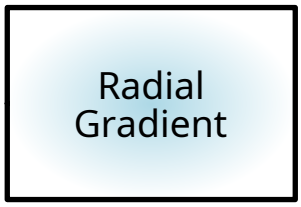
More advanced gradients may be added manually with *add_svgdef* to define the gradient.

```
with schemdraw.Drawing() as d:
    d.add_svgdef('''
```

(continues on next page)

(continued from previous page)

```
        <radialGradient id="mygrad" cx="50%" cy="50%" r="50%" fx="50%" fy="50%">
        <stop offset="0%" stop-color="lightblue" />
        <stop offset="100%" stop-color="white" />
        </radialGradient>''')
    flow.Box().fill('url(#mygrad)').label('Radial\nGradient')
```



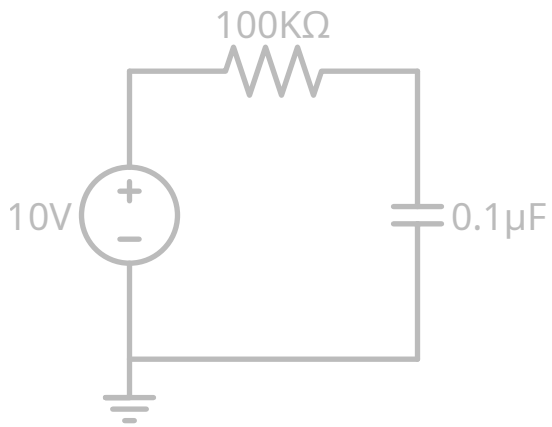
2.3.8 Themes

Schemdraw also supports themeing, to enable dark mode, for example. The defined themes match those in the Jupyter Themes package:

- default (black on white)
- dark (white on black)
- solarizedd
- solarizedl
- onedork
- oceans16
- monokai
- gruvboxl
- gruvboxd
- grade3
- chesterish

They are enabled using *schemdraw.theme()*:

```
schemdraw.theme('monokai')
with schemdraw.Drawing():
    elm.Resistor().label('100KΩ')
    elm.Capacitor().down().label('0.1μF', loc='bottom')
    elm.Line().left()
    elm.Ground()
    elm.SourceV().up().label('10V')
```



2.4 Backends

The backend is the "canvas" on which a schematic is drawn. Schemdraw supports two backends: Matplotlib, and SVG.

2.4.1 Matplotlib Backend

By default, all schematics are drawn on a Matplotlib axis if Matplotlib is installed and importable. A new Matplotlib Figure and Axis will be created, with no frame or borders. A schematic may be added to an existing Axis by using the *schemdraw.Drawing.draw()* method and setting the *canvas* parameter to an existing Axis instance.

The Matplotlib backend renders text labels as primative lines and arcs rather than text elements by default. This has the downside that SVG editors, such as Inkscape, cannot perform textual searches on the SVGs. The upside is that there is no dependence on installed fonts on the hosts that open the SVGs.

To configure Matplotlib to render labels as SVG text elements:

```
import matplotlib
matplotlib.rcParams['svg.fonttype'] = 'none'
```

2.4.2 SVG Backend

Schematics can also be drawn on directly to an SVG image backend. The SVG backend can be enabled for all drawings by calling:

```
schemdraw.use('svg')
```

The backend can be changed at any time. Alternatively, the backend can be set individually on each Drawing using the *canvas* parameter:

```
with schemdraw.Drawing(canvas='svg') as d:
    ...
```

Use additional Python libraries, such as pycairo, to convert the SVG output into other image formats.

Math Text

The SVG backend has basic math text support, including greek symbols, subscripts, and superscripts. However, if ziamath and latex2mathml packages are installed, they will be used for full Latex math support.

The SVG backend can produce searchable-text SVGs by setting:

```
schemdraw.svgconfig.text = 'text'
```

However, text mode does not support full Latex compatibility. To switch back to rendering text as SVG paths:

```
schemdraw.svgconfig.text = 'path'
```

Some SVG renderers are not fully compatible with SVG2.0. For better compatibility with SVG1.x, use

```
schemdraw.svgconfig.svg2 = False
```

The decimal precision of SVG elements can be set using

```
schemdraw.svgconfig.precision = 2
```

2.4.3 Backend Comparison

Reasons to choose the SVG backend include:

- No Matplotlib/Numpy dependency required (huge file size savings if bundling an executable).
- Speed. The SVG backend draws 4-10x faster than Matplotlib, depending on the circuit complexity.

Reasons to use Matplotlib backend:

- To customize the schematic after drawing it by using other Matplotlib functionality.
- To render directly in other, non-SVG, image formats, with no additional code.

CHAPTER

THREE

CIRCUIT ELEMENTS

3.1 Basic Elements

See *Electrical Elements* for complete class definitions for these elements.

3.1.1 Two-terminal

Two-terminal devices subclass `schemdraw.elements.Element2Term`, and have leads that will be extended to make the element the desired length depending on the arguments. All two-terminal elements define *start*, *end*, and *center* anchors for placing, and a few define other anchors as shown in blue in the tables below. Some elements have optional parameters, shown in parenthesis in the table below.

Resistors

Resistors elements change style based on IEEE/U.S. vs IEC/European style configured by `schemdraw.elements.style()`. Selectable elements, such as *Resistor*, point to either *ResistorIEEE* or *ResistorIEC*, for example.

IEEE Style

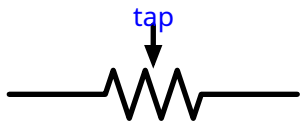
IEEE style, common in the U.S., is the default, or it can be configured using

```
elm.style(elm.STYLE_IEEE)
```

Resistor

ResistorVar

Potentiometer



Photoresistor

Fuse

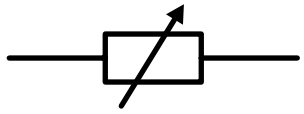
IEC/European Style

IEC style can be enabled using

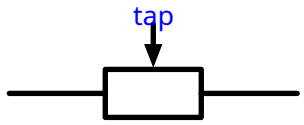
```
elm.style(elm.STYLE_IEC)
```

Resistor

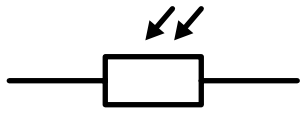
ResistorVar



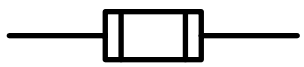
Potentiometer



Photoresistor



Fuse



All Resistors

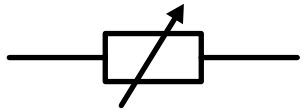
Either IEEE or IEC styles are always available with these Elements.

ResistorIEEE

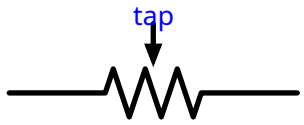
ResistorIEC

ResistorVarIEEE

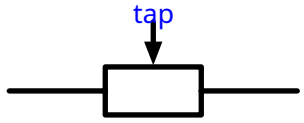
ResistorVarIEC



PotentiometerIEEE

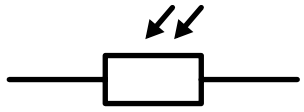


PotentiometerIEC



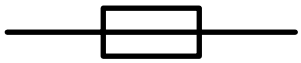
PhotoresistorIEEE

PhotoresistorIEC



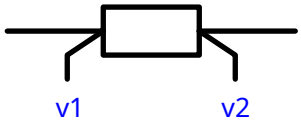
FuseUS

FuseIEEE



FuseIEC

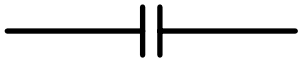
Rshunt



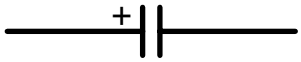
Capacitors and Inductors

Inductors have anchors intended for easy positioning of a mutual inductance dot near the element.

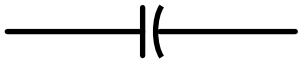
Capacitor



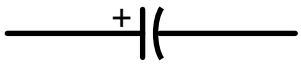
Capacitor(polar=True)



Capacitor2



Capacitor2(polar=True)

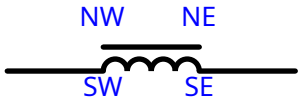


CapacitorVar

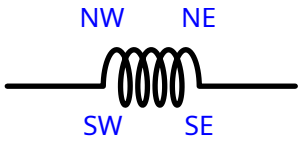
CapacitorTrim

Inductor

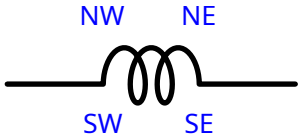
Inductor(core=True)



Inductor2



Inductor2(loops=2)



Diodes

All diodes may be filled by passing *fill=True*.

Diode

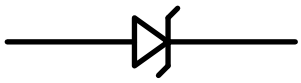
Diode(fill=True)

Schottky

DiodeTunnel

DiodeShockley

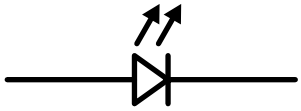
Zener



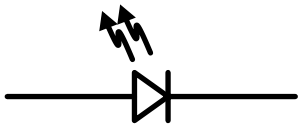
DiodeTVS

Varactor

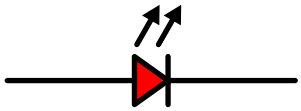
LED



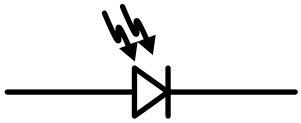
LED2



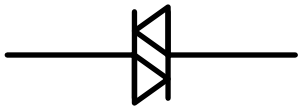
LED(fill='red')



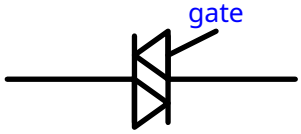
Photodiode



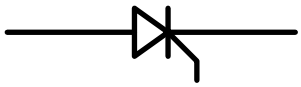
Diac



Triac



SCR



Miscellaneous

Breaker

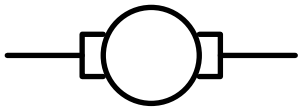
Crystal

CPE

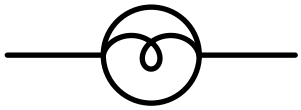
Josephson

Josephson(box=True)

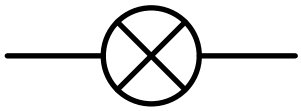
Motor



Lamp

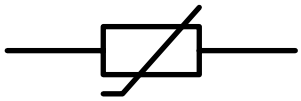


Lamp2



Neon

Thermistor

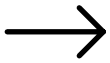


Memristor

Memristor2

Jack

Plug



Terminal

SparkGap

Nullator

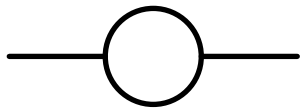
Norator

CurrentMirror

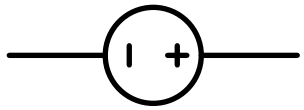
VoltageMirror

Sources and Meters

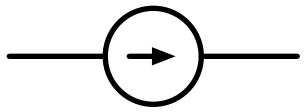
Source



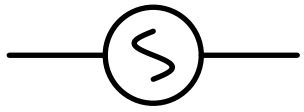
SourceV



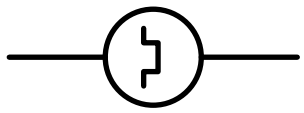
SourceI



SourceSin

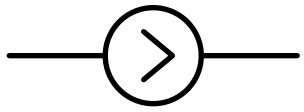


SourcePulse

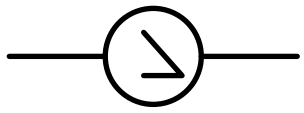


SourceSquare

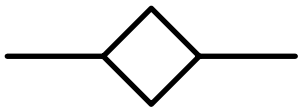
SourceTriangle



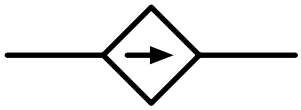
SourceRamp



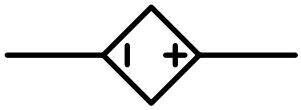
SourceControlled



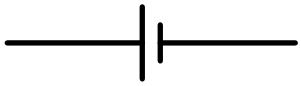
SourceControlledI



SourceControlledV



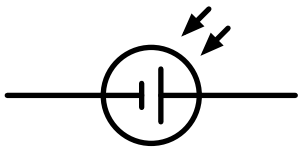
BatteryCell



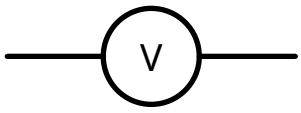
Battery

BatteryDouble

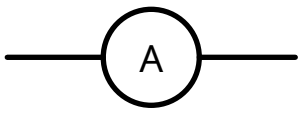
Solar



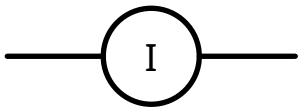
MeterV



MeterA

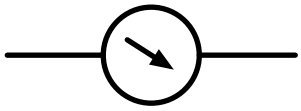


MeterI



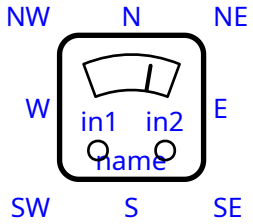
MeterOhm

MeterArrow

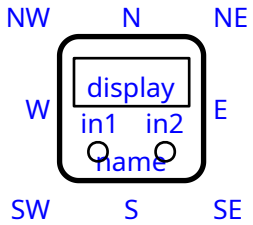


Box-Style Meters

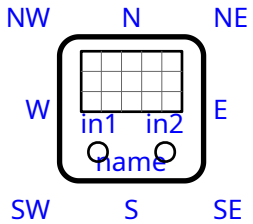
MeterAnalog



MeterDigital



Oscilloscope

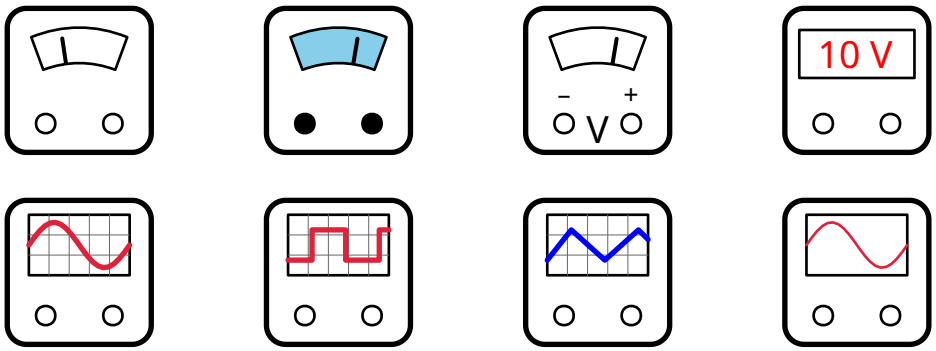


The "Box" style meters have many configuration options. See class definitions for more details. Some examples are shown below.

- *schemdraw.elements.sources.MeterAnalog*
- *schemdraw.elements.sources.MeterDigital*

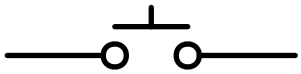
- *schemdraw.elements.sources.Oscilloscope*

```
with schemdraw.Drawing() as d:
    elm.MeterAnalog(needle_percent=30)
    d.move(2)
    elm.MeterAnalog(window_fill='skyblue', input_fill='black').anchor('in1')
    d.move(2)
    elm.MeterAnalog().label('V', loc='name').label('-', loc='in1', fontsize=10).label('+
↪', loc='in2', fontsize=10).anchor('in1')
    d.move(2)
    elm.MeterDigital().label('10 V', loc='display', color='red', fontsize=14).anchor('in1
↪')
    elm.Oscilloscope(signal='sine').at((0, -2))
    d.move(2)
    elm.Oscilloscope(signal='square').anchor('in1')
    d.move(2)
    elm.Oscilloscope(signal='triangle', signal_color='blue').anchor('in1')
    d.move(2)
    elm.Oscilloscope(signal='sine', signal_lw=1, grid=False).anchor('in1')
```

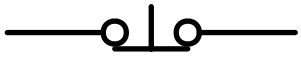


Switches

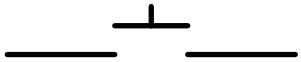
Button



Button(nc=True)



Button(contacts=False)

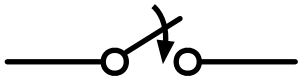


Switch

Switch(nc=True)

Switch(action='open')

Switch(action='close')



Switch(contacts=False)

Switch(contacts=False, nc=True)

SwitchReed

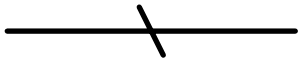
Lines and Arrows

Line

Arrow

Arrow(double=True)

DataBusLine



Also see *Connecting Elements*.

3.1.2 Single-Terminal

Single terminal elements are drawn about a single point, and do not move the current drawing position.

Power and Ground

Ground

GroundSignal

GroundChassis

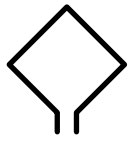
Vss

Vdd

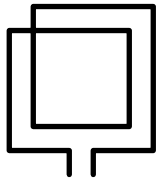
Antennas

Antenna

AntennaLoop



AntennaLoop2



Connection Dots

Dot

Dot(open=True)

DotDotDot

Arrowhead

NoConnect

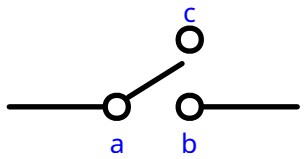
3.1.3 Switches

The standard toggle switch is listed with other two-terminal elements above. Other switch configurations are shown here.

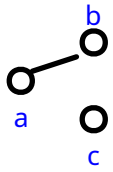
Single-pole double-throw

Two options for SPDT switches can be also be drawn with arrows by adding *action='open'* or *action='close'* parameters.

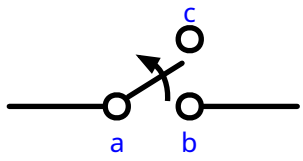
SwitchSpdt



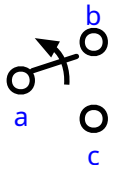
SwitchSpdt2



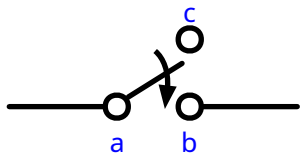
SwitchSpdt(action='open')



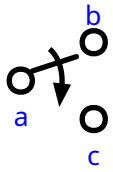
SwitchSpdt2(action='open')



SwitchSpdt(action='close')



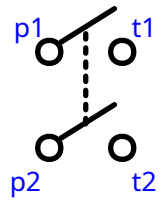
SwitchSpdt2(action='close')



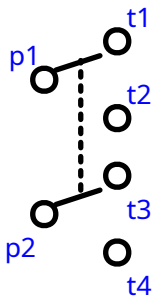
Double-pole

DPST and DPDT switches have a *link* parameter for disabling the dotted line lnking the poles.

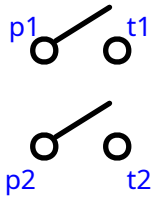
SwitchDpst



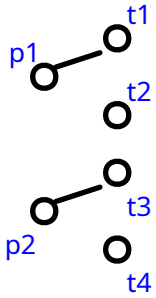
SwitchDpdt



SwitchDpst(link=False)



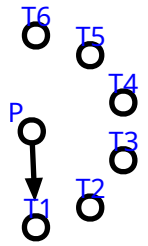
SwitchDpdt(link=False)



Rotary Switch

The rotary switch *schemdraw.elements.switches.SwitchRotary* takes several parameters, with *n* being the number of contacts and other parameters defining the contact placement.

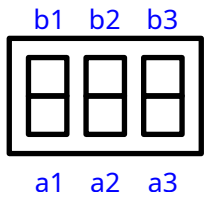
SwitchRotary(n=6)



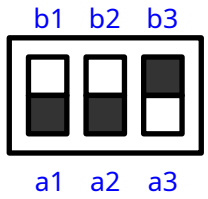
DIP Switch

A set of switches in a dual-inline package, where can show each switch flipped up or down. See *schemdraw.elements.switches.SwitchDIP* for options.

SwitchDIP



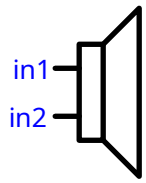
SwitchDIP(pattern=[0, 0, 1])



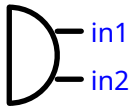
3.1.4 Audio Elements

Speakers, Microphones, Jacks

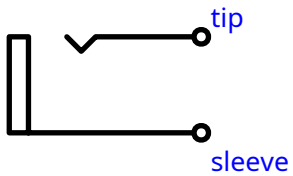
Speaker



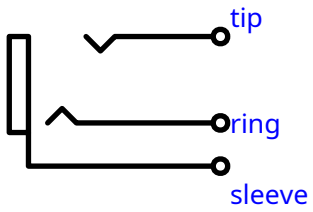
Mic



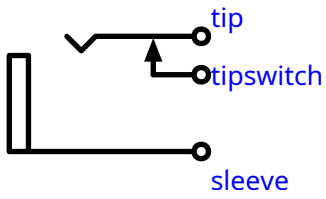
AudioJack



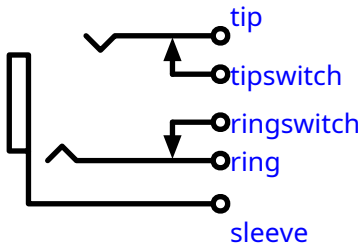
AudioJack(ring=True)



AudioJack(switch=True)



AudioJack(switch=True, ring=True, ringswitch=True)



3.1.5 Labels

The *Label* element can be used to add a label anywhere. The *Gap* is like an "invisible" element, useful for marking the voltage between output terminals.

Gap(label=['+', 'Gap', '-'])

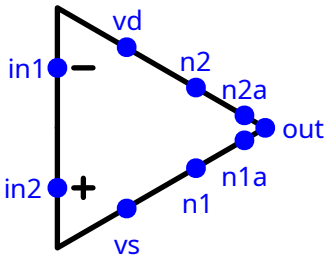
Label(label='Hello')

Tag(label='Tag')

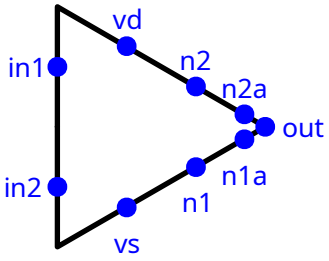
3.1.6 Operational Amplifiers

The *schemdraw.elements.opamp.Opamp* element defines several anchors for various inputs, including voltage supplies and offset nulls. Optional leads can be added using the *leads* parameter, with anchors exteded to the ends of the leads.

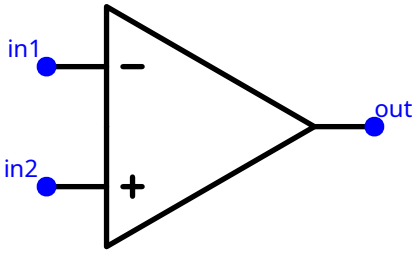
Opamp



Opamp(sign=False)



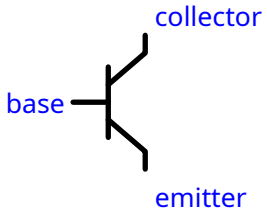
Opamp(leads=True)



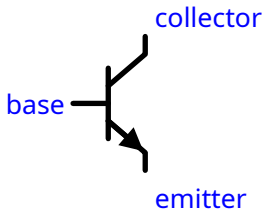
3.1.7 Transistors

Bipolar Junction Transistors

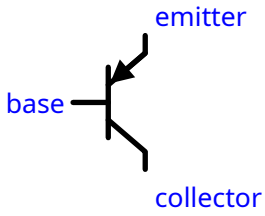
Bjt



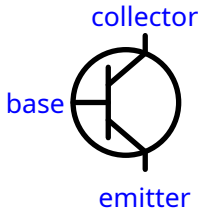
BjtNpn



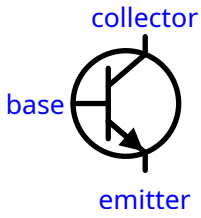
BjtPnp



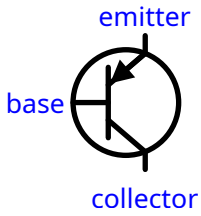
Bjt(circle=True)



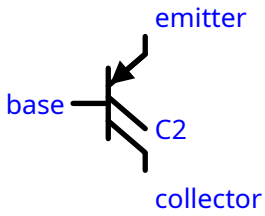
BjtNpn(circle=True)



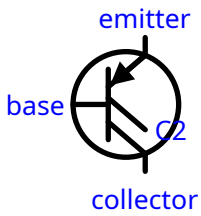
BjtPnp(circle=True)



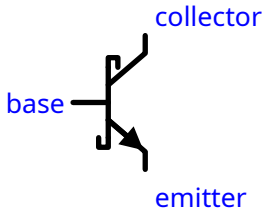
BjtPnp2c



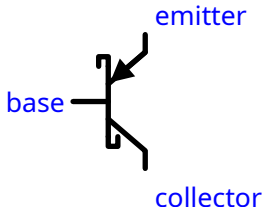
BjtPnp2c(circle=True)



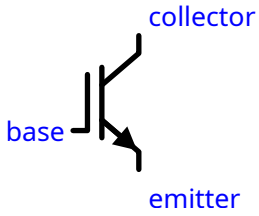
NpnSchottky



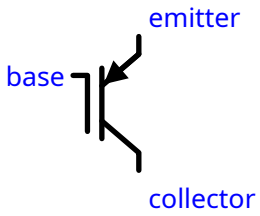
PnpSchottky



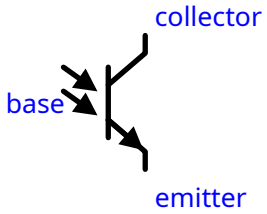
IgbtN



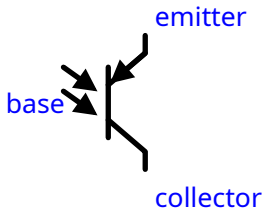
IgbtP



NpnPhoto

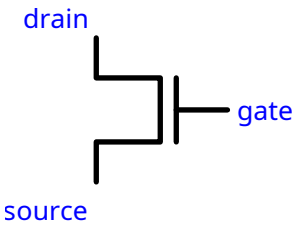


PnpPhoto

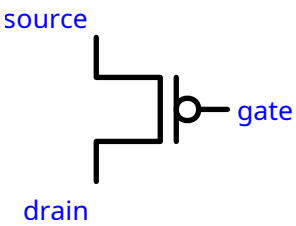


Field-Effect Transistors

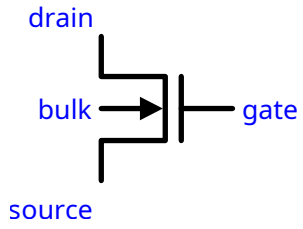
NFet



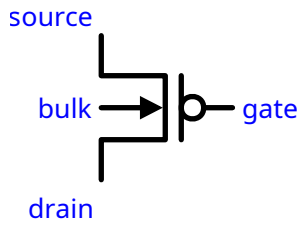
PFet



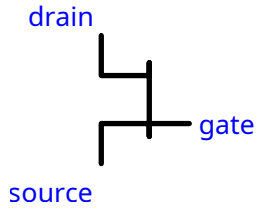
NFet(bulk=True)



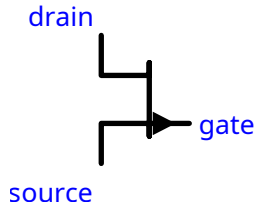
PFet(bulk=True)



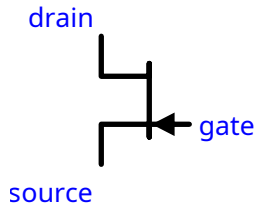
JFet



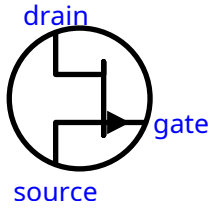
JFetN



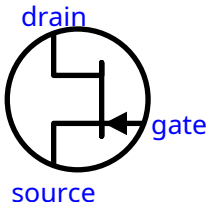
JFetP



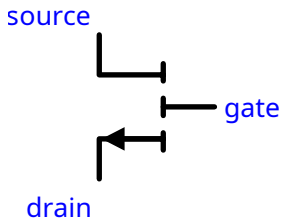
JFetN(circle=True)



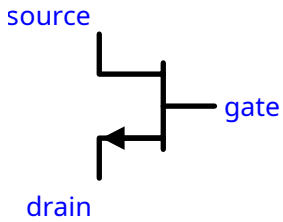
JFetP(circle=True)



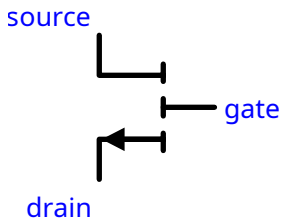
Hemt



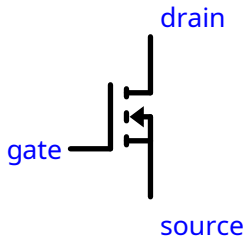
Hemt(split=False)



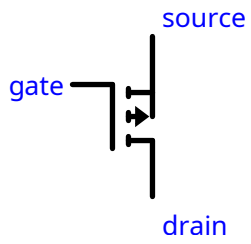
Hemt(arrow=False)



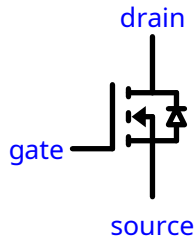
NMos



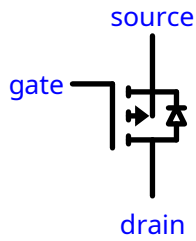
PMos



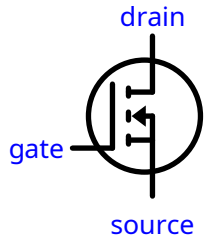
NMos(diode=True)



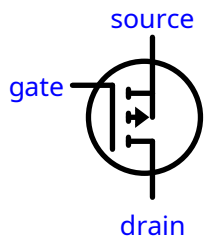
PMos(diode=True)



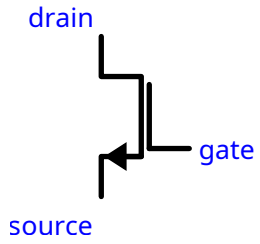
NMos(circle=True)



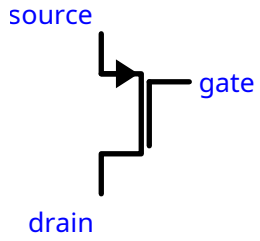
PMos(circle=True)



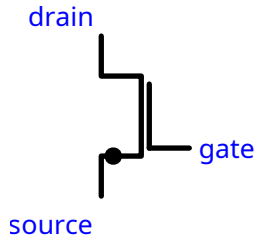
AnalogNFet



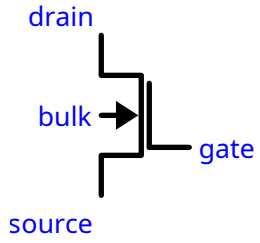
AnalogPFet



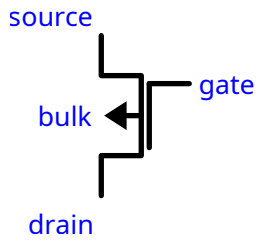
AnalogBiasedFet



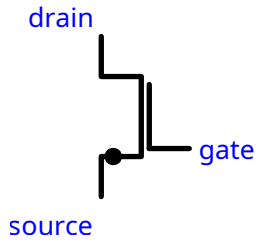
AnalogNFet(bulk=True)



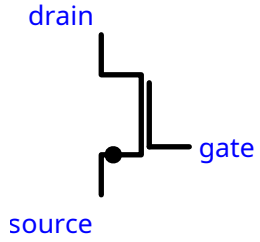
AnalogPFet(bulk=True)



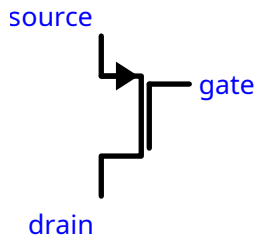
AnalogBiasedFet(bluk=True)



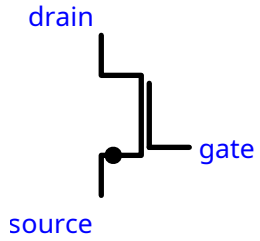
AnalogBiasedFet(arrow=False)



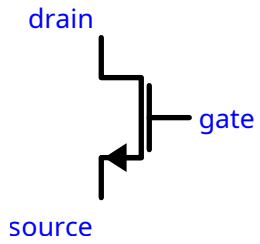
AnalogPFet(arrow=False)



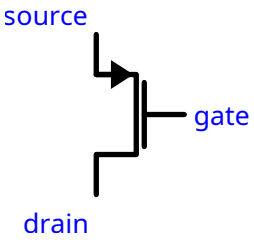
AnalogBiasedFet(arrow=False)



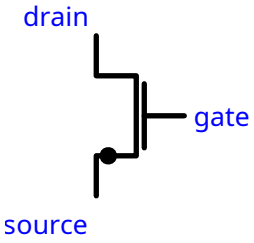
AnalogNFet(offset_gate=False)



AnalogPFet(offset_gate=False)



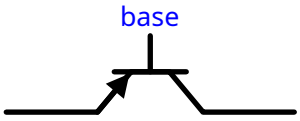
AnalogBiasedFet(offset_gate=False)



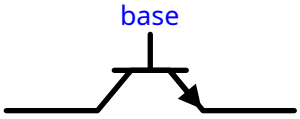
"Two-Terminal" Transistors

Another set of transistor elements subclass *schemdraw.elements.Element2Term* so they have emitter and collector (or source and drain) leads extended to the desired length. These can be easier to place centered between endpoints, for example.

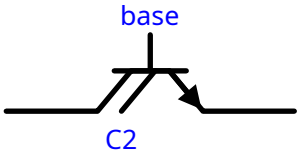
BjtNpn2



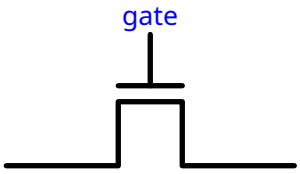
BjtPnp2



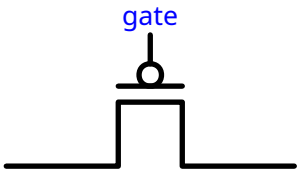
BjtPnp2c2



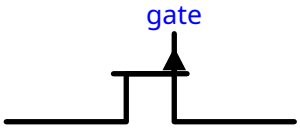
NFet2



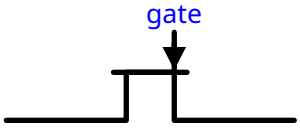
PFet2



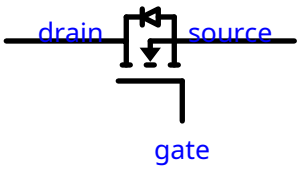
JFetN2



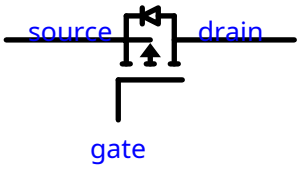
JFetP2



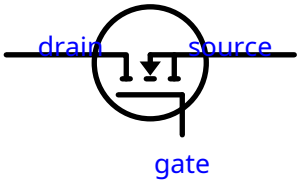
NMos2(diode=True)



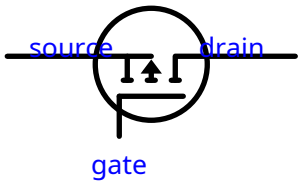
PMos2(diode=True)



NMos2(circle=True)



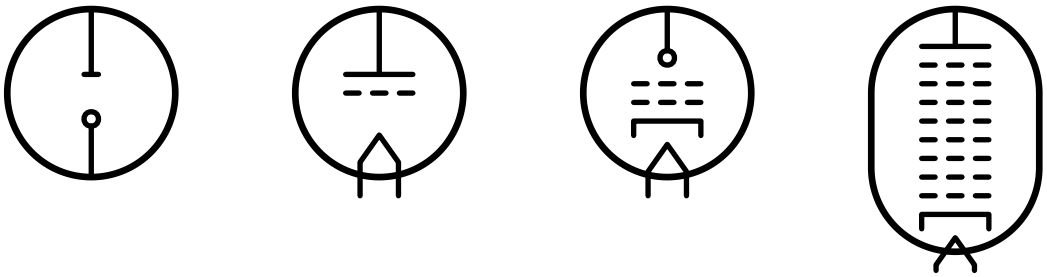
PMos2(circle=True)



3.1.8 Vacuum Tubes

A generic configurable vacuum tube is provided in *schemdraw.elements.tubes.VacuumTube*. It may be created with any number of grids, an indirectly-heated or cold cathode, multiple anode options, and an optional heater filament. Some options are below.

```
with schemdraw.Drawing():
    elm.VacuumTube(cathode='cold', anodetype='narrow', grids=0, heater=False)
    elm.VacuumTube(cathode='none', heater=True).at((3, 0))
    elm.VacuumTube(anodetype='dot', grids=2).at((6, 0))
    elm.VacuumTube(grids=8).at((9, 0))
```



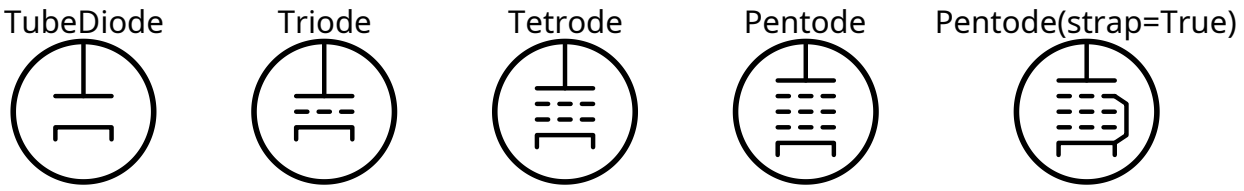
For the basic tube, anchors are *anode*, *cathode*, *cathode_R* (the right-hand connection to the cathode), *grid*, *grid_R*, and heater anchors *heat1* and *heat2*. When multiple grids are present, the grid anchors are *grid_N* and *grid_NR* where *N* is the grid number from the top down.

Common vacuum tubes are defined from the base *VacuumTube* element.

- *schemdraw.elements.tubes.TubeDiode*
- *schemdraw.elements.tubes.Triode*
- *schemdraw.elements.tubes.Tetrode*
- *schemdraw.elements.tubes.Pentode*

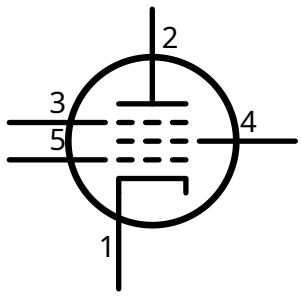
The triode, tetrode, and pentode include anchors named for function: *control*, *screen*, and *suppressor*. The pentode includes a *strap* parameter to connect the suppressor to the cathode.

```
with schemdraw.Drawing():
    elm.TubeDiode().label('TubeDiode')
    elm.Triode().at((3, 0)).label('Triode')
    elm.Tetrode().at((6, 0)).label('Tetrode')
    elm.Pentode().at((9, 0)).label('Pentode')
    elm.Pentode(strap=True).at((12.5, 0)).label('Pentode(strap=True)')
```



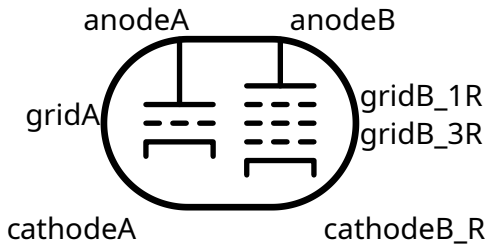
The anchors are positioned internal to the tube envelope to allow connection from either side. Connecting lines extend into the envelope, and pin numbers appear just outside.

```
with schemdraw.Drawing():
    tube = (elm.Pentode()
                .label('1', 'cathode')
                .label('2', 'anode')
                .label('3', 'suppressor')
                .label('4', 'screen_R')
                .label('5', 'control')
            )
    elm.Line().down().at(tube.cathode).length(1)
    elm.Line().up().at(tube.anode).length(.5)
    elm.Line().left().at(tube.suppressor).length(1)
    elm.Line().right().at(tube.screen_R).length(1)
    elm.Line().left().at(tube.control).length(1)
```



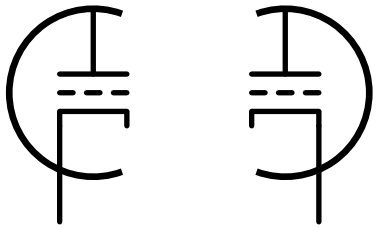
A dual vacuum tube may be drawn, with independent number of grids on each side. Anchors are designated *A* on the left element and *B* on the right element. *schemdraw.elements.tubes.DualVacuumTube*

```
with schemdraw.Drawing():
    t = (elm.DualVacuumTube(grids_left=1, grids_right=3, heater=False)
                .label('anodeA', 'anodeA', halign='right')
                .label('anodeB', 'anodeB')
                .label('cathodeA', 'cathodeA')
                .label('cathodeB_R', 'cathodeB_R', halign='left')
                .label('gridA', 'gridA', valign='center')
                .label('gridB_1R', 'gridB_1R', valign='center')
                .label('gridB_3R', 'gridB_3R', valign='center')
            )
```



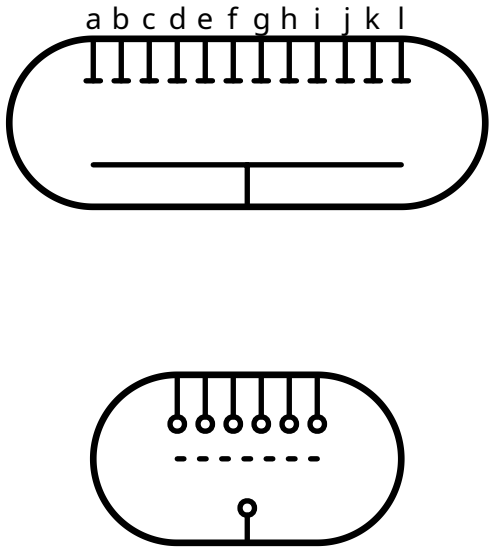
Sometimes, multiple tubes in one envelope are drawn in a split style. This may be enabled using *split='left'* or *split='right`*.

```
with schemdraw.Drawing():
    tube1 = elm.Triode(split='left')
    elm.Line().down().at(tube1.cathode).length(1)
    tube2 = elm.Triode(split='right').at((2, 0))
    elm.Line().down().at(tube2.cathode_R).length(1)
```



Nixie Tubes are multi-anode tubes drawn with *schemdraw.elements.tubes.NixieTube*. The cathode may be *cold*, *T* or *none* and anodetype may be *narrow* or *dot* or *none*.

```
with schemdraw.Drawing():
    t = elm.NixieTube(anodes=12)
    for i in range(12):
        t.label(chr(ord('a')+i), f'anode{i}', halign='center', ofst=(0, .1))

    elm.NixieTube(anodes=6, anodetype='dot', cathode='cold', grid=True).at((0, -3.5))
```

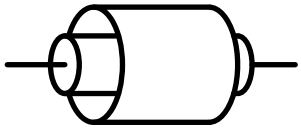


3.1.9 Cables

schemdraw.elements.cables.Coax and *schemdraw.elements.cables.Triax* cables are 2-Terminal elements that can be made with several options and anchors. Coax parameters include length, radius, and leadlen for setting the distance between leads and the shell. Triax parameters include length, radiusinner, radiusouter, leadlen, and shieldoffstart for offseting the outer shield from the inner guard.

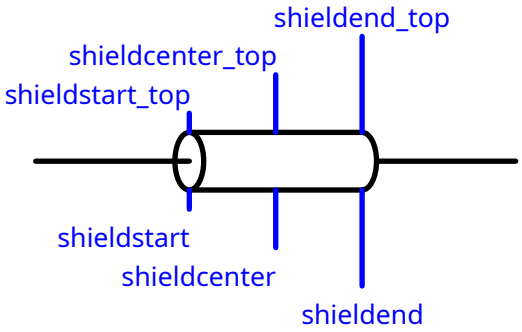
Coax

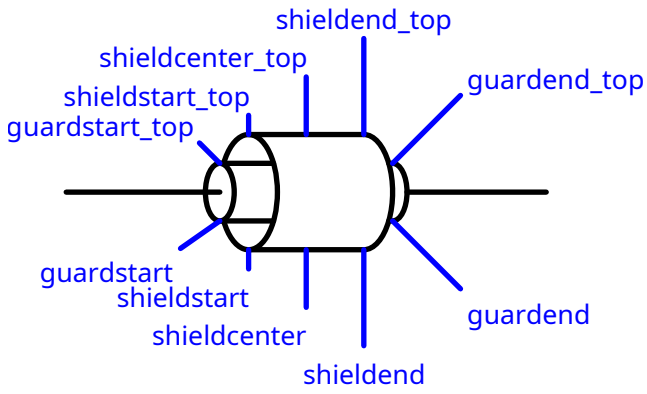
Triax()



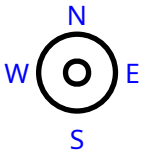
Coax(length=5, radius=0.5)

Triax(length=5, radiusinner=0.5)





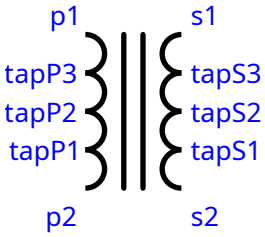
CoaxConnect



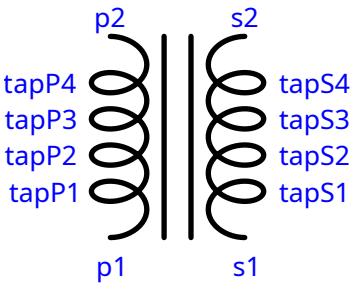
3.1.10 Transformers

The *schemdraw.elements.xform.Transformer* element is used to create various transformers. The number of turns on each side is set with the *t1* and *t2* parameters. As integers, they define the total number of turns on that side. Multi-phase transformers may be created by setting *t1* or *t2* to a tuple of turns for each phase.

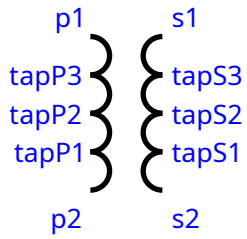
Transformer



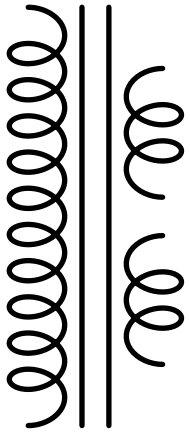
Transformer(loop=True)



Transformer(core=False)




```
with schemdraw.Drawing():
    elm.Transformer(t1=10, t2=(2, 2), loop=True)
```



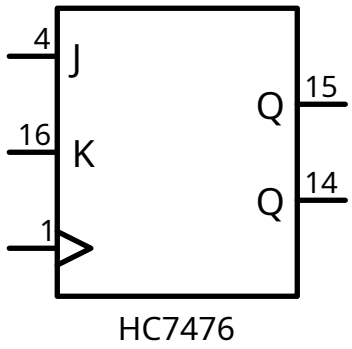
3.2 Integrated Circuits

The *schemdraw.elements.intcircuits.Ic* class is used to make integrated circuits, multiplexers, and other black box elements. The *schemdraw.elements.intcircuits.IcPin* class is used to define each input/output pin before adding it to the Ic.

All pins will be given an anchor name of *inXY* where X is the side (L, R, T, B), and Y is the pin number along that side. Pins also define anchors based on the *name* parameter. If the *anchorname* parameter is provided for the pin, this name will be used, so that the pin *name* can be any string even if it cannot be used as a Python variable name.

Here, a J-K flip flop, as part of an HC7476 integrated circuit, is drawn with input names and pin numbers.

```
JK = elm.Ic(pins=[elm.IcPin(name='>', pin='1', side='left'),
                  elm.IcPin(name='K', pin='16', side='left'),
                  elm.IcPin(name='J', pin='4', side='left'),
                  elm.IcPin(name=r'$\overline{Q}$', pin='14', side='right', anchorname=
↪'QBAR'),
                  elm.IcPin(name='Q', pin='15', side='right')],
            edgepadW = .5,  # Make it a bit wider
            pinspacing=1).label('HC7476', 'bottom', fontsize=12)
display(JK)
```

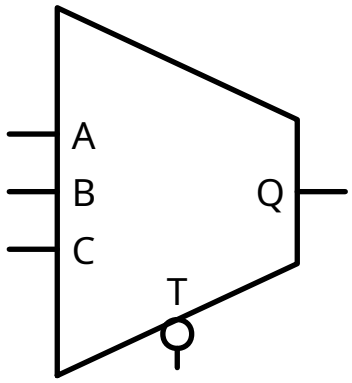


Notice the use of *$overline{Q}$* to acheive the label on the inverting output. The anchor positions can be accessed using attributes, such as *JK.Q* for the non-inverting output. However, inverting output is named *$overline{Q}*, which is not accessible using the typical dot notation. It could be accessed using *getattr(JK, r'$overline{Q}$')*, but to avoid this an alternative anchorname of *QBAR* was defined.

3.2.1 Multiplexers

Multiplexers and demultiplexers are drawn with the *schemdraw.elements.intcircuits.Multiplexer* class which wraps the Ic class.

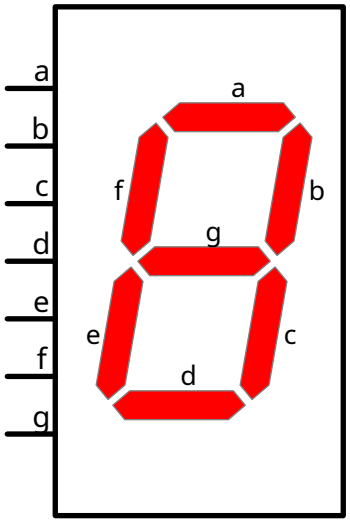
```
elm.Multiplexer(
    pins=[elm.IcPin(name='C', side='L'),
          elm.IcPin(name='B', side='L'),
          elm.IcPin(name='A', side='L'),
          elm.IcPin(name='Q', side='R'),
          elm.IcPin(name='T', side='B', invert=True)],
    edgepadH=-.5)
```



See the *Circuit Gallery* for more examples.

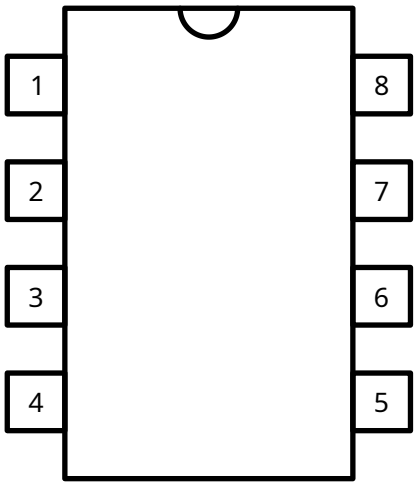
3.2.2 Seven-Segment Display

A seven-segment display, in *schemdraw.elements.intcircuits.SevenSegment*, provides a single digit with several options including decimal point and common anode or common cathode mode. The *schemdraw.elements.intcircuits.sevensegdigit()* method generates a list of Segment objects that can be used to add a digit to another element, for example to make a multi-digit display.



3.2.3 DIP Integrated Circuits

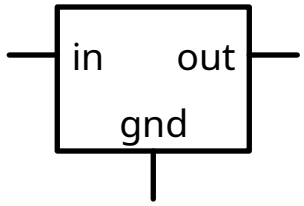
Integrated circuits can be drawn in dual-inline package style with *schemdraw.elements.intcircuits.IcDIP*. Anchors allow connecting elements externally to show the IC in a circuit, or interanally to show the internal configuration of the IC (see *741 Opamp, DIP Layout*.)



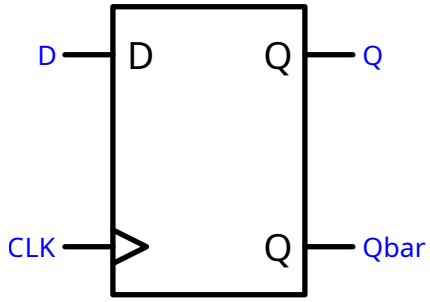
3.2.4 Predefined ICs

A few common integrated circuits are predefined as shown below.

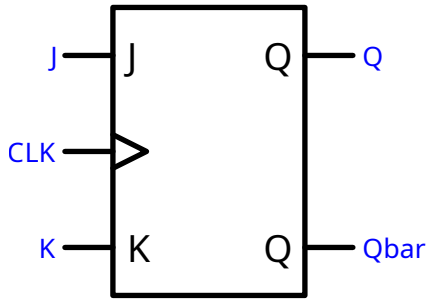
VoltageRegulator



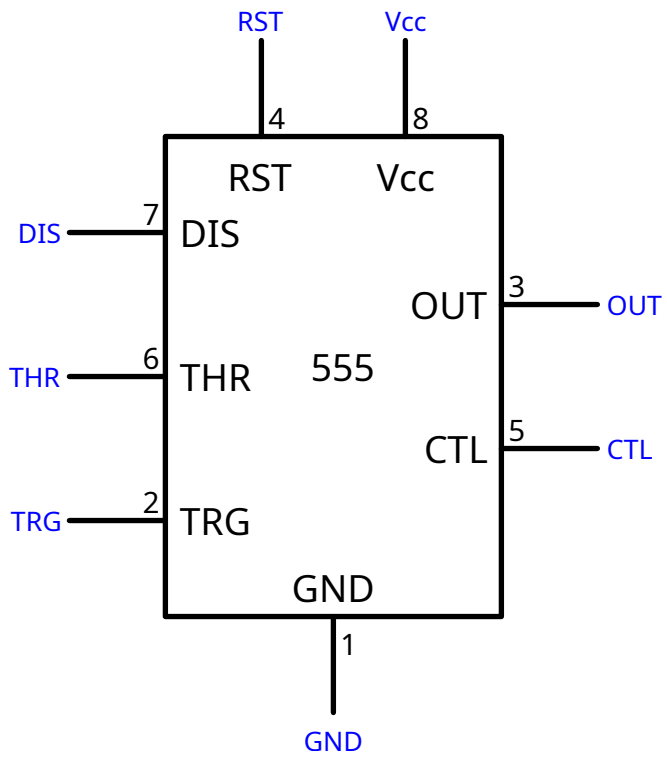
DFlipFlop



JKFlipFlop



Ic555



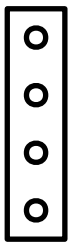
3.3 Connectors

All connectors are defined with a default pin spacing of 0.6, matching the default pin spacing of the *schemdraw.elements.intcircuits.Ic* class, for easy connection of multiple signals.

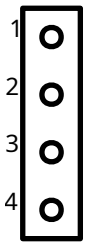
3.3.1 Headers

A *schemdraw.elements.connectors.Header* is a generic Header block with any number of rows and columns. It can have round, square, or screw-head connection points.

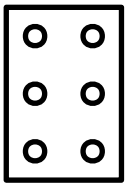
Header



Header(shownumber=True)

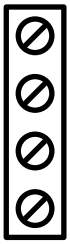


Header(rows=3, cols=2)

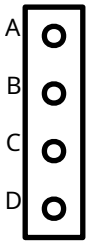


Header(style='square')

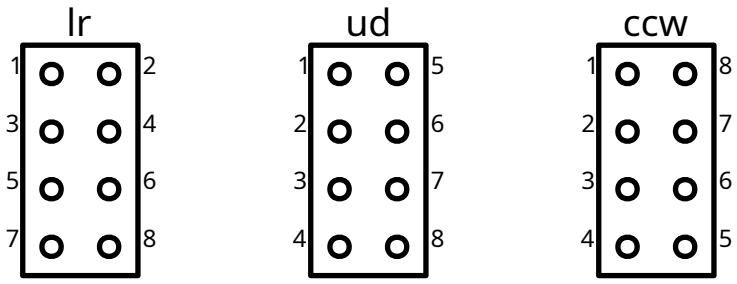
Header(style='screw')



Header(pinsleft=['A', 'B', 'C', 'D'])

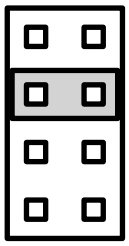


Header pins are given anchor names *pin1*, *pin2*, etc. Pin number labels and anchor names can be ordered left-to-right (*lr*), up-to-down (*ud*), or counterclockwise (*ccw*) like a traditional IC, depending on the *numbering* argument. The *flip* argument can be set True to put pin 1 at the bottom.



A *schemdraw.elements.connectors.Jumper* element is also defined, as a simple rectangle, for easy placing onto a header.

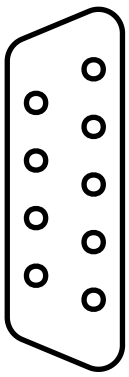
```
with schemdraw.Drawing():
    J = elm.Header(cols=2, style='square')
    elm.Jumper().at(J.pin3).fill('lightgray')
```



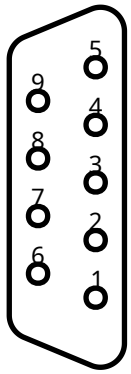
3.3.2 D-Sub Connectors

D-subminiature connectors DE9 (aliased to the commonly used name *DB9*), DB25, DA15, DC37, and DD50 are defined with anchors *pin1* through *pinX* where *X* is the number of pins in the connector. Classes are *schemdraw.elements.connectors.DE9*, *schemdraw.elements.connectors.DB25*, *schemdraw.elements.connectors.DA15*, *schemdraw.elements.connectors.DC37*, *schemdraw.elements.connectors.DD50*.

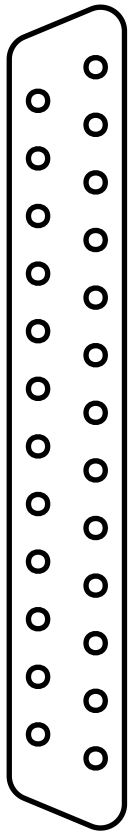
DE9



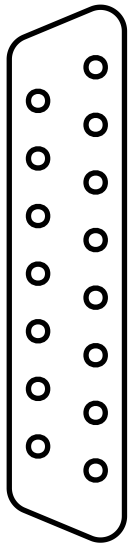
DE9(number=True)



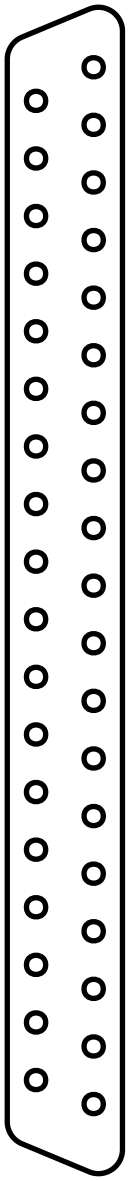
DB25



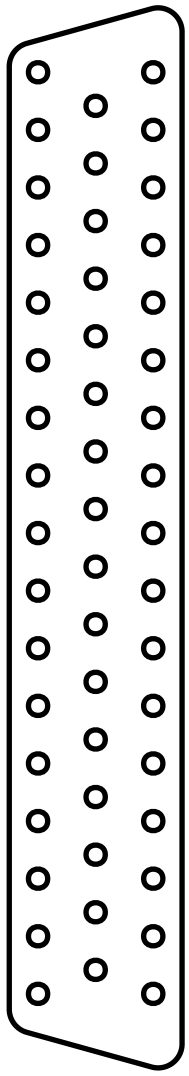
DA15



DC37



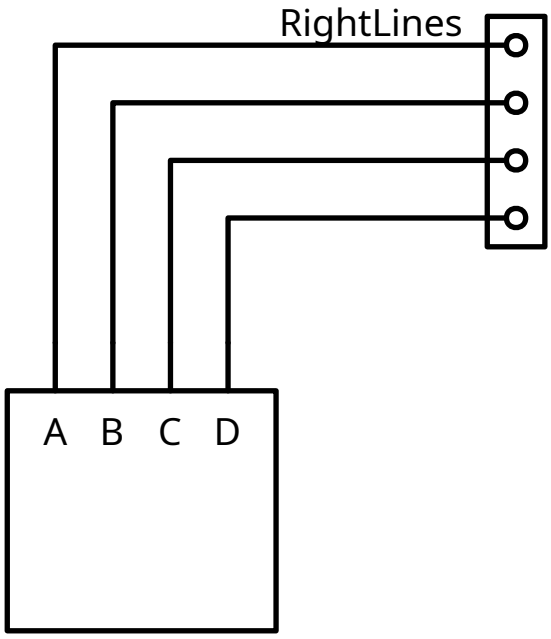
DD50



3.3.3 Multiple Lines

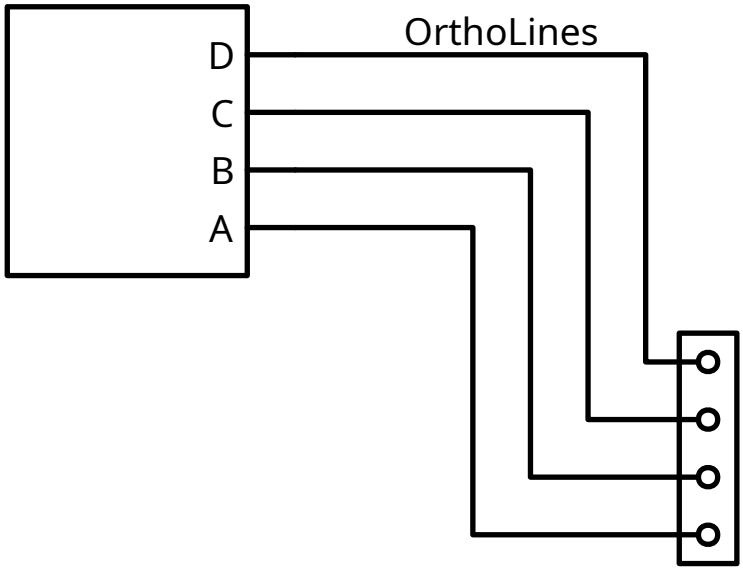
The *schemdraw.elements.connectors.RightLines* and *schemdraw.elements.connectors.OrthoLines* elements are useful for connecting multiple pins of an integrated circuit or header all at once. Both need an *at* and *to* location specified, along with the *n* parameter for setting the number of lines to draw. Use RightLines when the Headers are perpindicular to each other.

```
with schemdraw.Drawing():
    D1 = elm.Ic(pins=[elm.IcPin(name='A', side='t', slot='1/4'),
                      elm.IcPin(name='B', side='t', slot='2/4'),
                      elm.IcPin(name='C', side='t', slot='3/4'),
                      elm.IcPin(name='D', side='t', slot='4/4')])
    D2 = elm.Header(rows=4).at((5,4))
    elm.RightLines(n=4).at(D2.pin1).to(D1.D).label('RightLines')
```



OrthoLines draw a z-shaped orthogonal connection. Use OrthoLines when the Headers are parallel but vertically offset. Use the *xstart* parameter, between 0 and 1, to specify the position where the first OrthoLine turns vertical.

```
with schemdraw.Drawing():
    D1 = elm.Ic(pins=[elm.IcPin(name='A', side='r', slot='1/4'),
                      elm.IcPin(name='B', side='r', slot='2/4'),
                      elm.IcPin(name='C', side='r', slot='3/4'),
                      elm.IcPin(name='D', side='r', slot='4/4')])
    D2 = elm.Header(rows=4).at((7, -3))
    elm.OrthoLines(n=4).at(D1.D).to(D2.pin1).label('OrthoLines')
```

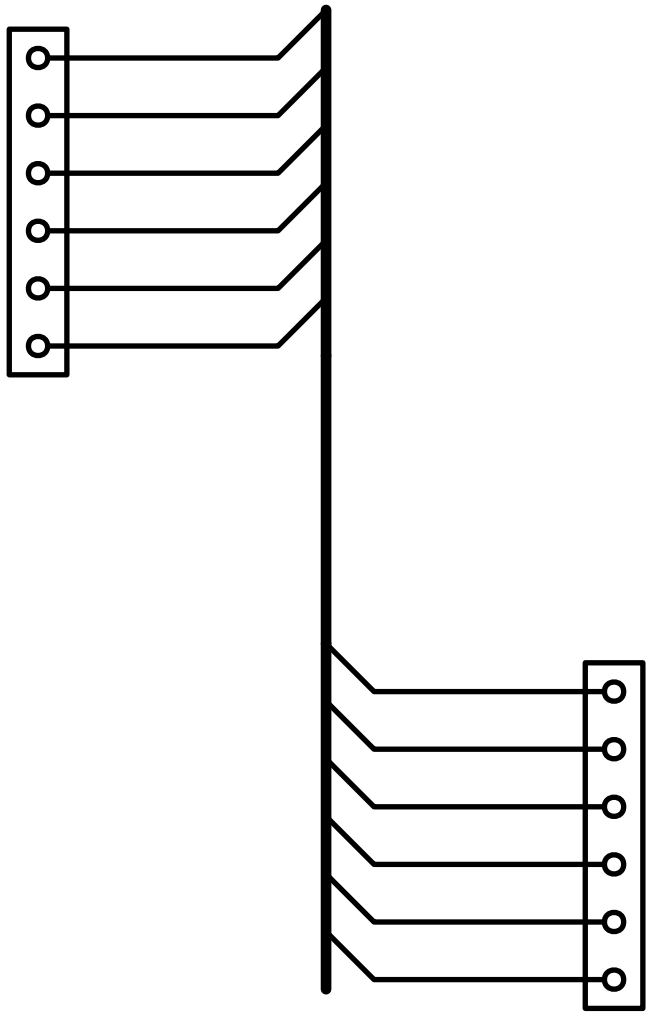


3.3.4 Data Busses

Sometimes, multiple I/O pins to an integrated circuit are lumped together into a data bus. The connections to a bus can be drawn using the *schemdraw.elements.connectors.BusConnect* element, which takes *n* the number of data lines and an argument. *schemdraw.elements.connectors.BusLine* is simply a wider line used to extend the full bus to its destination.

BusConnect elements define anchors *start*, *end* on the endpoints of the wide bus line, and *pin1*, *pin2*, etc. for the individual signals.

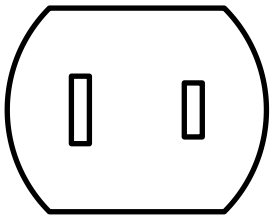
```
with schemdraw.Drawing():
    J = elm.Header(rows=6)
    B = elm.BusConnect(n=6).at(J.pin1)
    elm.BusLine().down().at(B.end).length(3)
    B2 = elm.BusConnect(n=6).anchor('start').reverse()
    elm.Header(rows=6).at(B2.pin1).anchor('pin1')
```



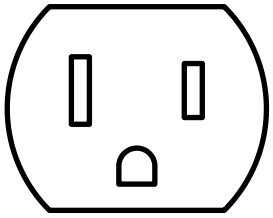
3.3.5 Outlets

Power outlets and plugs are drawn using *OutletX* classes, with international styles A through L. Each has anchors *hot*, *neutral*, and *ground* (if applicable). The *plug* parameter fills the prongs to indicate a plug versus an outlet.

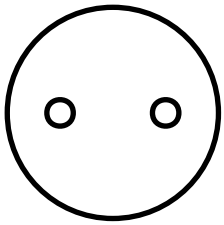
OutletA



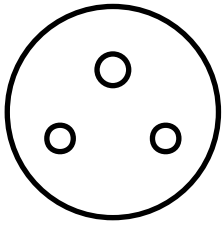
OutletB



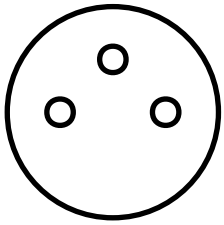
OutletC



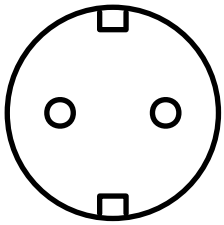
OutletD



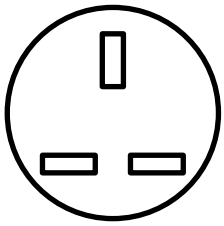
OutletE



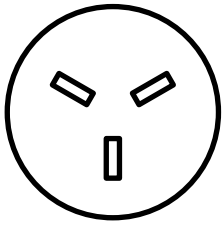
OutletF



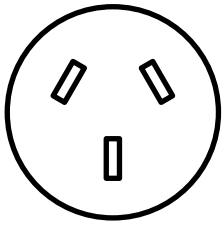
OutletG



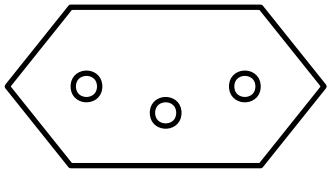
OutletH



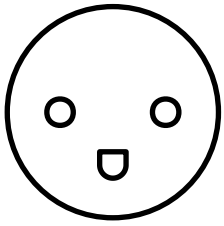
OutletI



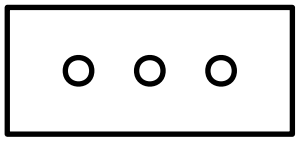
OutletJ



OutletK



OutletL



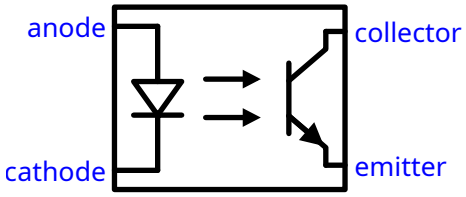
3.4 Compound Elements

Several compound elements defined based on other basic elements.

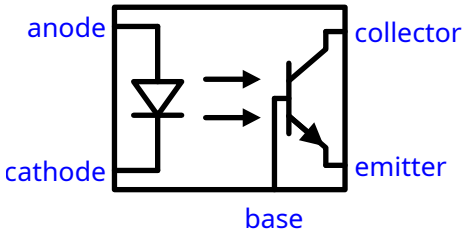
3.4.1 Optocoupler

schemdraw.elements.compound.Optocoupler can be drawn with or without a base contact.

Optocoupler



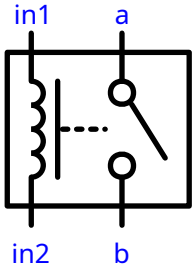
Optocoupler(base=True)



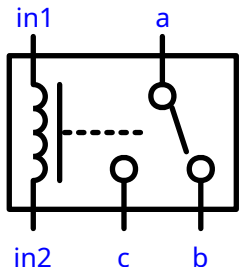
3.4.2 Relay

schemdraw.elements.compound.Relay can be drawn with different options for switches and inductor solenoids.

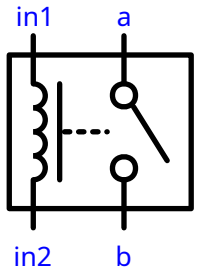
Relay



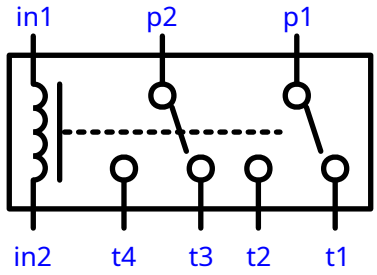
Relay(switch='spdt')



Relay(swithc='dpst')



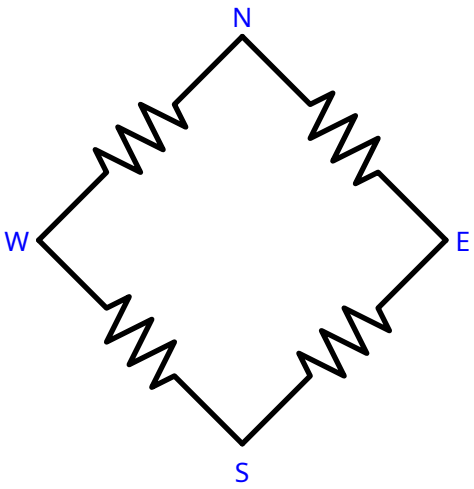
Relay(switch='dpdt')



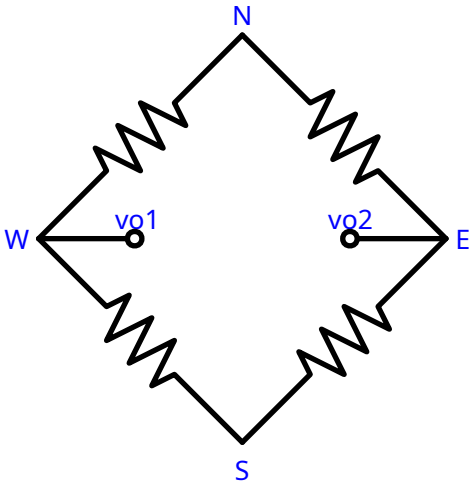
3.4.3 Wheatstone

schemdraw.elements.compound.Wheatstone can be drawn with or without the output voltage taps. The *labels* argument specifies a list of labels for each resistor.

Wheatstone



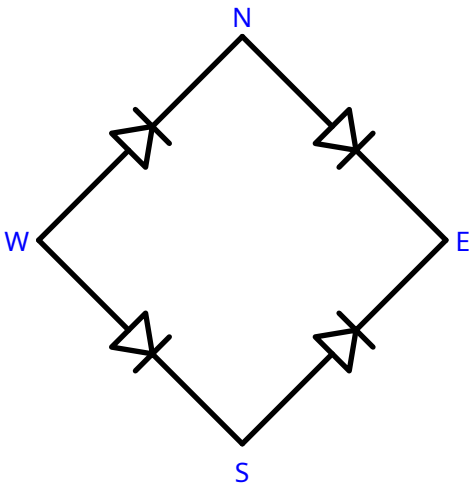
Wheatstone(vout=True)



3.4.4 Rectifier

`schemdraw.elements.compound.Rectifier` draws four diodes at 45 degree angles. The *labels* argument specifies a list of labels for each diode.

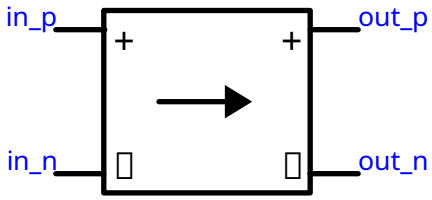
Rectifier



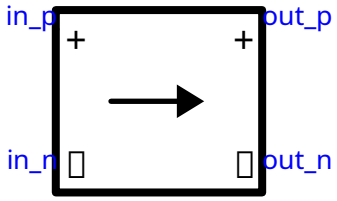
3.4.5 Two-ports

Twoport elements share the interface defined by *schemdraw.elements.twoports.ElementTwoport*, providing a set of anchors and various styling options. The terminals and box can be enabled or disabled using the *terminals* and *box* arguments. In addition, the *boxfill*, *boxlw*, and *boxls* provide the option to style the outline separately from other elements.

TwoPort

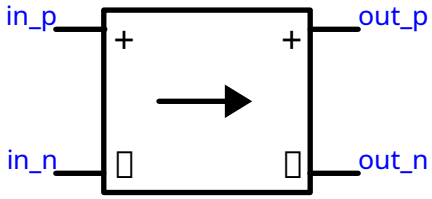


TwoPort(terminals=False, boxlw=3)

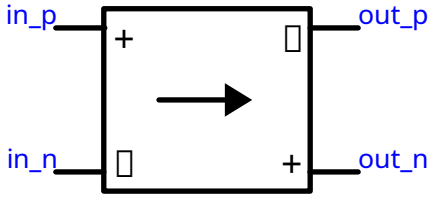


Generic

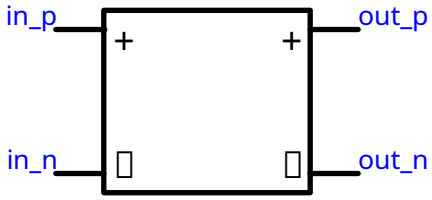
TwoPort



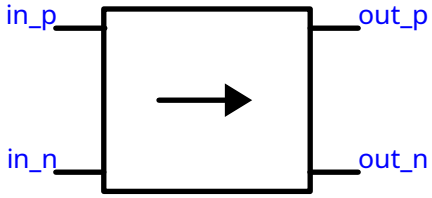
TwoPort(reverse_output=True)



TwoPort(arrow=False)



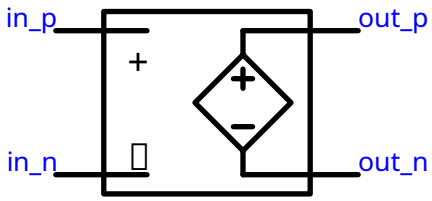
TwoPort(sign=False)



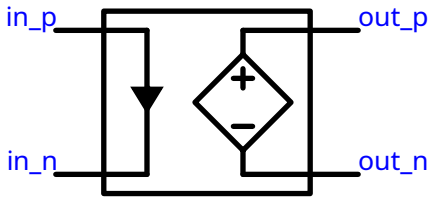
Transactors (ideal amplifiers)

Like the generic twoport, the transactors provide the option to reverse the direction of the output or current using the *reverse_output* argument.

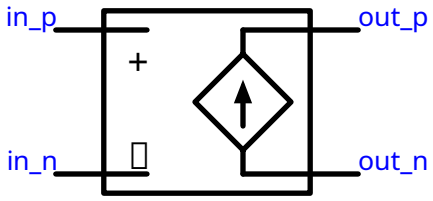
VoltageTransactor



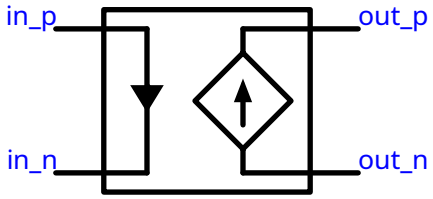
TransimpedanceTransactor



TransadmittanceTransactor

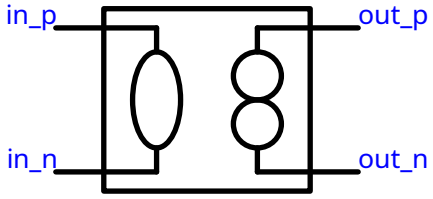


CurrentTransactor

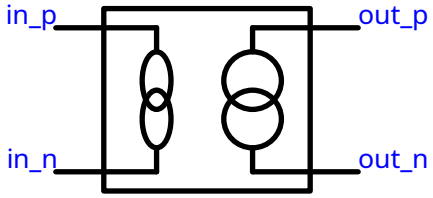


Pathological

Nullor



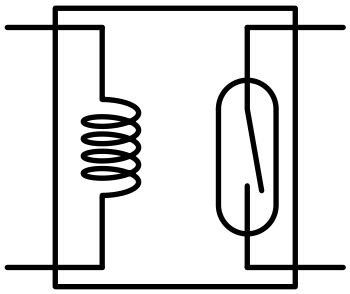
VMCMPair



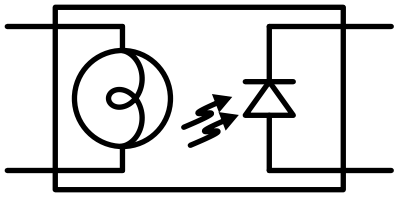
Custom

The *schemdraw.elements.twoports.ElementTwoport* class can be used to define custom twoports by specifying an *input_element* and *output_element*. The *bpadx*, *bpady*, *minw*, *unit*, *width* can be used to tune the horizontal and vertical padding, minimum width of the elements, length of components, and width of the twoport respectively.

```
elm.ElementTwoport(
    input_element=elm.Inductor2,
    output_element=elm.SwitchReed,
    unit=2.5, width=2.5)
```



```
elm.ElementTwoport(
    input_element=elm.Lamp,
    output_element=partial(elm.Photodiode, reverse=True, flip=True),
    width=3)
```



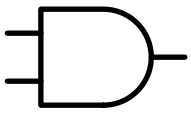
3.5 Digital Logic

Logic gates can be drawn by importing the *schemdraw.logic.logic* module:

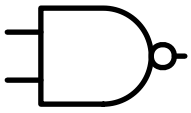
```
from schemdraw import logic
```

Logic gates are shown below. Gates define anchors for *out* and *in1*, *in2*, etc. *Buf*, *Not*, and *NotNot*, and their Schmitt-trigger counterparts, are two-terminal elements that extend leads.

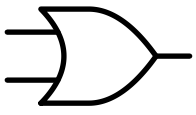
And



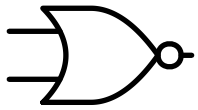
Nand



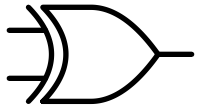
Or



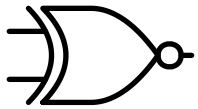
Nor



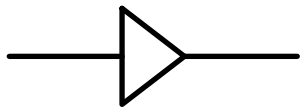
Xor



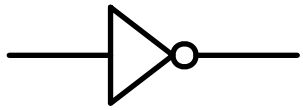
Xnor



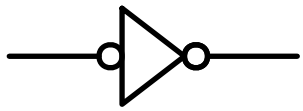
Buf



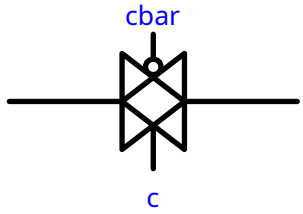
Not



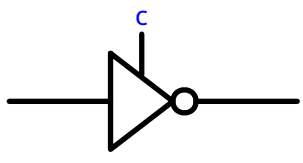
NotNot



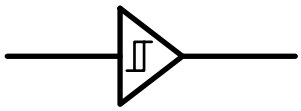
Tgate



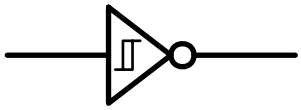
Tristate



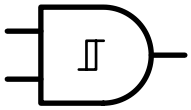
Schmitt



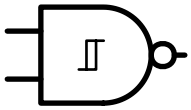
SchmittNot



SchmittAnd

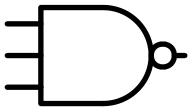


SchmittNand

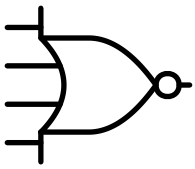


Gates with more than 2 inputs can be created using the *inputs* parameter. With more than 3 inputs, the back of the gate will extend up and down.

Nand(inputs=3)

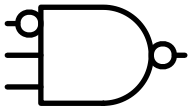


Nor(inputs=4)



Finally, any input can be pre-inverted (active low) using the *inputnots* keyword with a list of input numbers, starting at 1 to match the anchor names, on which to add an invert bubble.

Nand(inputs=3, inputnots=[1])

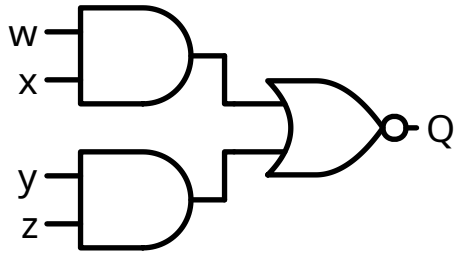


3.5.1 Logic Parser

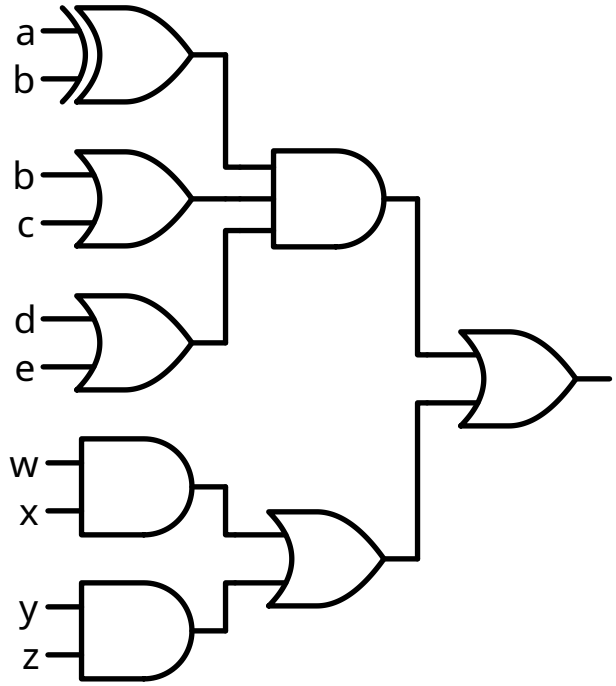
Logic trees can also be created from a string logic expression such as "(a and b) or c" using using *schemdraw.parsing.logic_parser.logicparse()*. The logic parser requires the pyparsing module.

Examples:

```
from schemdraw.parsing import logicparse
logicparse('not ((w and x) or (y and z))', outlabel=r'$\overline{Q}$')
```

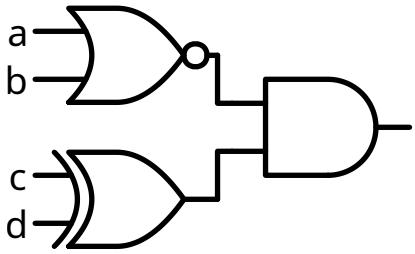



```
logicparse('((a xor b) and (b or c) and (d or e)) or ((w and x) or (y and z))')
```



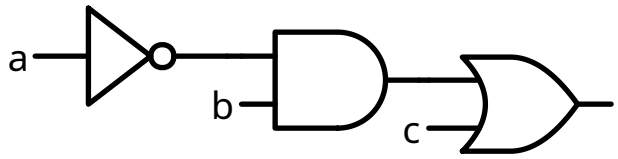
Logicparse understands spelled-out logic functions "and", "or", "nand", "nor", "xor", "xnor", "not", but also common symbols such as "+", "&", "⊕" representing "or", "and", and "xor".

```
logicparse('¬ (a ∨ b) & (c ⊻ d)')  # Using symbols
```



Use the *gateH* and *gateW* parameters to adjust how gates line up:

```
logicparse('(not a) and b or c', gateH=.5)
```



3.5.2 Truth Tables

Simple tables can be drawn using the *schemdraw.logic.table.Table* class. This class is included in the logic module as its primary purpose was for drawing logical truth tables.

The tables are defined using typical Markdown syntax. The *colfmt* parameter works like the LaTeX tabular environment parameter for defining lines to draw between table columns: "cc|c" draws three centered columns, with a vertical line before the last column. Each column must be specified with a 'c', 'r', or 'l' for center, right, or left justification Two pipes (||), or a double pipe character (‖) draw a double bar between columns. Row lines are added to the table string itself, with either — or === in the row.

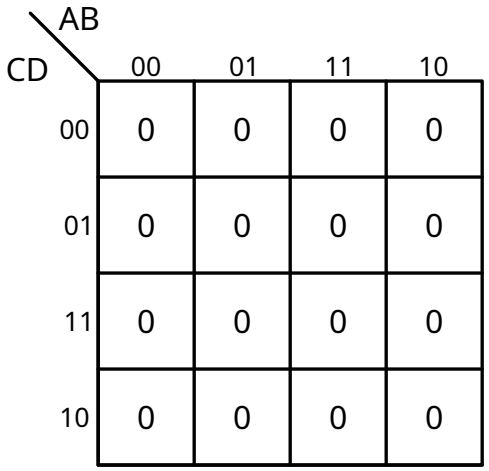
```
table = '''
 A | B | C
---|---|---
 0 | 0 | 0
 0 | 1 | 0
 1 | 0 | 0
 1 | 1 | 1
'''
logic.Table(table, colfmt='cc||c')
```

A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

3.5.3 Karnaugh Maps

Karnaugh Maps, or K-Maps, are useful for simplifying a logical truth table into the smallest number of gates. Schemdraw can draw K-Maps, with 2, 3, or 4 input variables, using the *schemdraw.logic.kmap.Kmap* class.

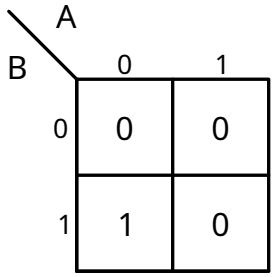
```
logic.Kmap(names='ABCD')
```



The *names* parameter must be a string with 2, 3, or 4 characters, each defining the name of one input variable. The *truthtable* parameter contains a list of tuples defining the logic values to display in the map. The first *len(names)* elements are 0's and 1's defining the position of the cell, and the last element is the string to display in that cell. The *default* parameter is a string to show in each cell of the K-Map when that cell is undefined in the *truthtable*.

For example, this 2x2 K-Map has a '1' in the 01 position, and 0's elsewhere:

```
logic.Kmap(names='AB', truthtable=[('01', '1')])
```



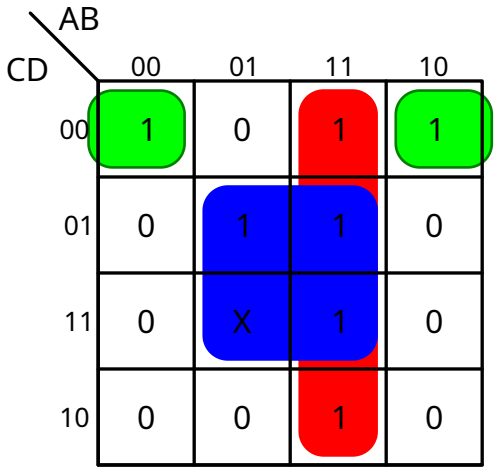
K-Maps are typically used by grouping sets of 1's together. These groupings can be drawn using the *groups* parameter. The keys of the *groups* dictionary define which cells to group together, and the values of the dictionary define style parameters for the circle around the group. Each key must be a string of length *len(names)*, with either a *0*, *1*, or . in each position. As an example, with *names='ABCD'*, a group key of *"1..."* will place a circle around all cells where A=1. Or *".00."* draws a circle around all cells where B and C are both 0. Groups will automatically "wrap" around the edges. Parameters of the style dictionary include *color*, *fill*, *lw*, and *ls*.

```
logic.Kmap(names='ABCD',
           truthtable=[('1100', '1'),
                       ('1101', '1'),
                       ('1111', '1'),
                       ('1110', '1'),
```

(continues on next page)

(continued from previous page)

```
                    ('0101', '1'),
                    ('0111', 'X'),
                    ('1101', '1'),
                    ('1111', '1'),
                    ('0000', '1'),
                    ('1000', '1')],
          groups={'11..': {'color': 'red', 'fill': '#ff000033'},
                  '.1.1': {'color': 'blue', 'fill': '#0000ff33'},
                  '.000': {'color': 'green', 'fill': '#00ff0033'}})
```



Note

Kmap and *Table* are both Elements, meaning they may be added to a schemdraw *Drawing* with other schematic components. To save a standalone *Kmap* or *Table* to an image file, first add it to a drawing, and save the drawing:

```
with schemdraw.Drawing(file='truthtable.svg'):
    logic.Table(table, colfmt='cc||c')
```

3.6 Timing Diagrams

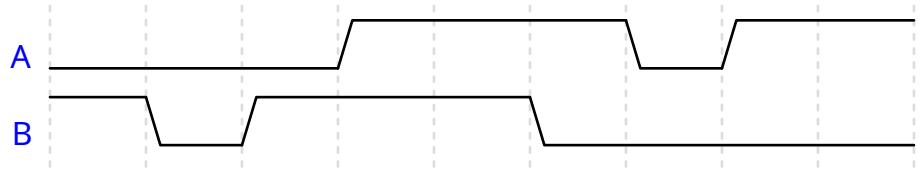
Digital timing diagrams may be drawn using the *schemdraw.logic.timing.TimingDiagram* Element in the `schemdraw.logic` module. Bit Field diagrams may be drawn using *schemdraw.logic.bitfield.BitField*.

Timing diagrams and bit fields are set up using the WaveJSON syntax used by the WaveDrom JavaScript application. Schemdraw adds a number of additional parameters to extend the configurability of WaveDrom.

3.6.1 Timing Diagrams

```
from schemdraw import logic
```

```
logic.TimingDiagram(
    {'signal': [
        {'name': 'A', 'wave': '0..1..01.'},
        {'name': 'B', 'wave': '101..0...'}]})
```



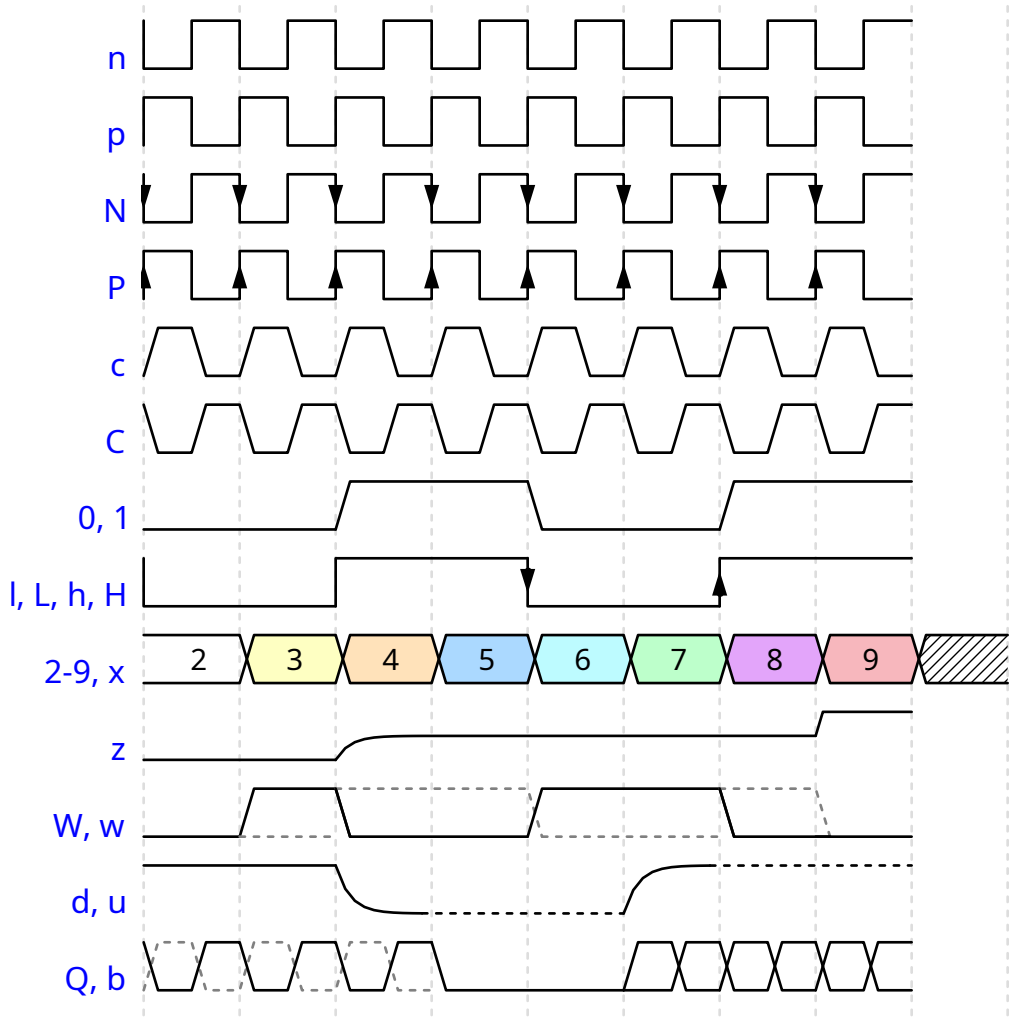
The input is a dictionary containing a *signal*, which is a list of each wave to show in the diagram. Each signal is a dictionary which typically contains a *name* and *wave*. An empty dictionary leaves a blank row in the diagram.

Every character in the *wave* specifies the state of the wave for one period. For example, the '1' and '0' in the above wave strings indicate a logic high and low signal. The dot . means the previous state is repeated.

Waveform Types

All the waveform types are shown below.

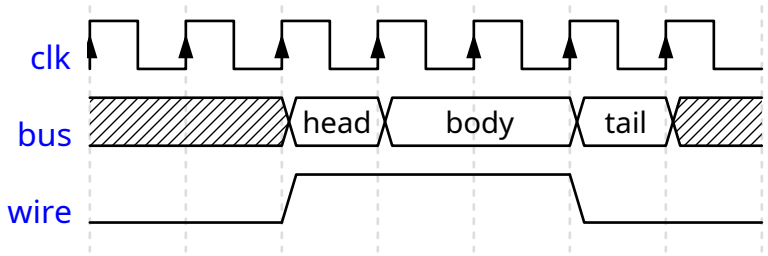
```
logic.TimingDiagram(
    {'signal':[
        {'name': 'n', 'wave': 'n.......'},
        {'name': 'p', 'wave': 'p.......'},
        {'name': 'N', 'wave': 'N.......'},
        {'name': 'P', 'wave': 'P.......'},
        {'name': 'c', 'wave': 'c.......'},
        {'name': 'C', 'wave': 'C.......'},
        {'name': '0, 1', 'wave': '0.1.0.1.'},
        {'name': 'l, L, h, H', 'wave': 'l.h.L.H.'},
        {'name': '2-9, x', 'wave': '23456789x', 'data': '2 3 4 5 6 7 8 9 x'},
        {'name': 'z', 'wave': '0.z....1'},
        {'name': 'W, w', 'wave': '0wW.w.W0'},
        {'name': 'd, u', 'wave': '1.d..u..'},
        {'name': 'Q, b', 'wave': 'Q..0.bbb'},
    ]}
)
```


n
p
N
P
c
C
0, 1
l, L, h, H
2-9, x
2
3
4
5
6
7
8
9
z
W, w
d, u
Q, b

Wave Type	Description
n	Clock signal
n	Clock signal
p	Inverted clock signal
N	Clock signal with arrows
P	Inverted clock signal with arrows
c, C[1]	Clock signals with rise time
0	Low state, with rise time
1	High state, with rise time
l	Low state, no rise time
h	High state, no rise time
L	Low state with arrow
H	High state with arrow
2-9	Data states, with different colors
x	Invalid or don't-care data state
z	High impedance state, halfway up
d	Low state with pull-down curve
u	High state with pull-up curve
Q, q[1]	Differential clock
W, w[1]	Differential signal
b[1]	Half-period bit state
e[1]	Empty state

Example:

```
logic.TimingDiagram(
    {'signal': [
        {'name': 'clk', 'wave': 'P......'},
        {'name': 'bus', 'wave': 'x.==.=x', 'data': ['head', 'body', 'tail']},
        {'name': 'wire', 'wave': '0.1..0.'}]})
```



Signal Parameters

In addition to *wave*, each signal row has a number of parameters.

- **name**: Text to show to the left of the signal
- **data: Value to display inside data wave types. May be:**
  - List of strings, one item per data block
  - Space separated string

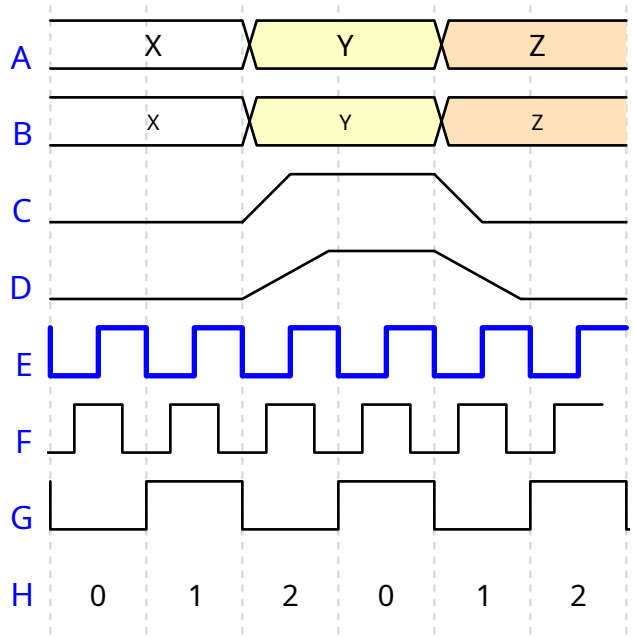
[1] Schemdraw extension to the original WaveDrom format

  - String in the format *{0, 1}*, where the values 0 and 1 are repeated through the signal[Page 96, 1]

- **node**: Define nodes for drawing arrows. See *Nodes and Edges*.
- **nodealign**: Alignment of nodes, either *signal* or *clock*. See *Nodes and Edges*.[Page 96, 1]
- **phase**: Introduce horizontal phase shift, as fraction of the period.
- **risetime**: Rise/fall time for signal types 0-9 and x in drawing units (default 0.1).[Page 96, 1]
- **period**: Change the period of the signal (default: 1)
- **level**: String with same length as *wave* defining the maximum y-value. See *Variable Voltage Levels*.[Page 96, 1]
- **color**: Color of the wave
- **lw**: Line width of the wave[Page 96, 1]
- **fontsize**: Fontsize of data labels[Page 96, 1]

Examples of some parameters are shown below. Note 'wave' may be omitted to only display data text.

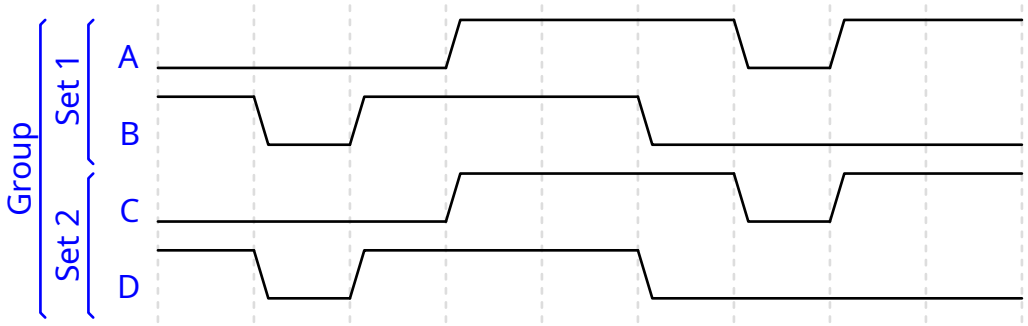
```
logic.TimingDiagram(
    {'signal':[
        {'name': 'A', 'wave': '2.3.4.', 'data': 'X Y Z'},
        {'name': 'B', 'wave': '2.3.4.', 'data': 'X Y Z', 'fontsize': 8},
        {'name': 'C', 'wave': '0.1.0.', 'risetime': 0.5},
        {'name': 'D', 'wave': '0.1.0.', 'risetime': 0.9},
        {'name': 'E', 'wave': 'n.....', 'color': 'blue', 'lw': 2},
        {'name': 'F', 'wave': 'n.....', 'phase': 0.25},
        {'name': 'G', 'wave': 'n.....', 'period': 2},
        {'name': 'H', 'data': '{0, 1, 2}'},
    ]}
)
```



Signal Groups

Signals may also be nested into different groups:

```
logic.TimingDiagram(
    {'signal': ['Group',
      ['Set 1',
        {'name': 'A', 'wave': '0..1..01.'},
        {'name': 'B', 'wave': '101..0...'}],
      ['Set 2',
        {'name': 'C', 'wave': '0..1..01.'},
        {'name': 'D', 'wave': '101..0...'}]
              ]})
```

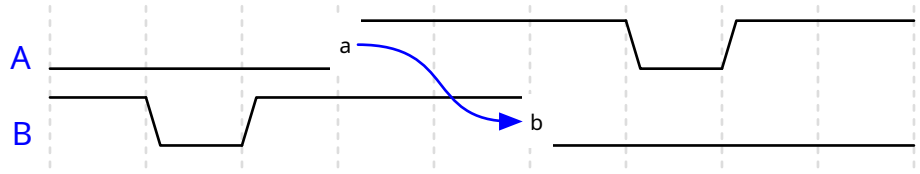


Nodes and Edges

The *node* keyword may be added to signals to define edge positions and labels within the wave. Within the node string, a lowercase letter indicates a node to be drawn on the wave. An uppercase letter defines a position without being drawn. A period leaves the position undefined.

Once nodes are defined, the *edge* parameter in the top-level dictionary provides a way to annotate transitions between different edges. The *edge* parameter is a list of strings, each defining an annotation. Each string starts and ends with a node letter, with a line type between. For example, "a->b" draws an arrow pointing from node a to b, and "c-d" draws a straight line from node c to d.

```
logic.TimingDiagram(
    {'signal': [
        {'name': 'A', 'wave': '0..1..01.', 'node': '...a.....'},
        {'name': 'B', 'wave': '101..0...', 'node': '.....b...'}],
     'edge': ['a~>b']
    })
```



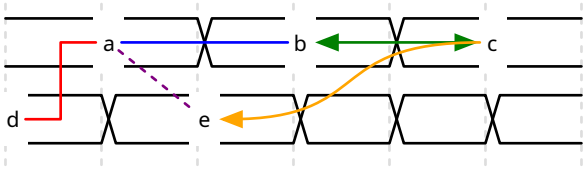
Line and arrow types include:

- **-**: Straight line
- **->** or **<-** or **<->**: Straight line with arrow heads

- **|-**: Vertical then horizontal straight line
- **-|**: Horizontal then vertical straight line
- **-|-**: "Z" shape line
- **~** or **-~** or **~-**: Various curved lines. May include arrow heads with '>' or '<'.

Colors may be specified placing the color in braces after the edge string. A linestyle of ':' or '–' may be added after the color, separated by a comma. Edge types are illustrated below.

```
logic.TimingDiagram(
    {'signal':[
        {'wave': '222222', 'node': '.a.b.c'},
        {'wave': '222222', 'node': 'd.e...'},
    ],
    'edge': [
        'a-b',
        'b<->c{green}',
        'd-|-a{red}',
        'a-e{purple,:}',
        'e<~c{orange}',
    ]}
)
```



Extended Edge Notation

Schemdraw adds additional "edge" string notations for more complex labeling of edge timings, including asynchronous start and end times and labels just above or below a wave[Page 96, 1].

Each edge string using this syntax takes the form

```
'[WaveNum:Period]<->[WaveNum:Period]{color,ls} Label'
```

Everything after the first space will be drawn as the label in the center of the line. The values in square brackets designate the start and end position of the line. *WaveNum* is the integer row number (starting at 0) of the wave, and *Period* is the possibly fractional number of periods in time for the node. *WaveNum* may be appended by a ^ or *v* to designate notations just above, or just below, the wave, respectively.

Between the two square-bracket expressions is the standard line/arrow type designator. In optional curly braces, the line color and linestyle may be entered.

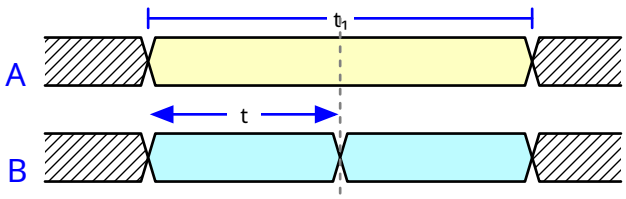
Some examples are shown here:

```
logic.TimingDiagram(
    {'signal': [
        {'name': 'A', 'wave': 'x3...x'},
        {'name': 'B', 'wave': 'x6.6.x'}],
     'edge': ['[0^:1]+[0^:5] $t_1$',
```

(continues on next page)

(continued from previous page)

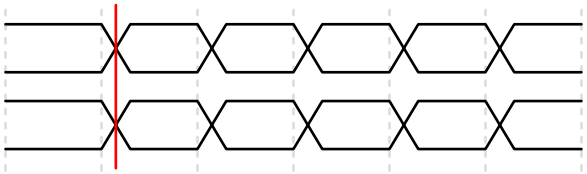
```
                '[1^:1]<->[1^:3] $t_o$',
                '[0^:3]-[1v:3]{gray,:}',
            ]},
    ygap=.5, grid=False)
```



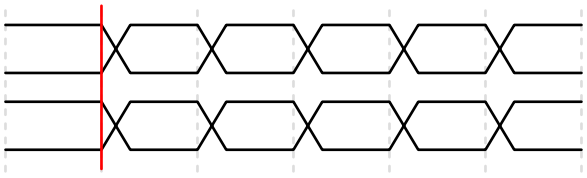
When placing edge labels above or below the wave, it can be useful to add the *ygap* parameter to TimingDiagram to increase the spacing between waves.

By default, nodes are aligned with the end of the risetime of a signal. To shift the node to the clock edge, set the *nodealign* parameter to 'clock'. The following examples show the difference.

```
logic.TimingDiagram(
    {'signal':[
        {'wave': '222222'},
        {'wave': '222222'},
    ],
    'edge': [
        '[0^:1]-[1v:1]{red}',
    ]},
    risetime=.3
)
```



```
logic.TimingDiagram(
    {'signal':[
        {'wave': '222222'},
        {'wave': '222222'},
    ],
    'edge': [
        '[0^:1]-[1v:1]{red}',
    ]},
    risetime=.3, nodealign='clock'
)
```



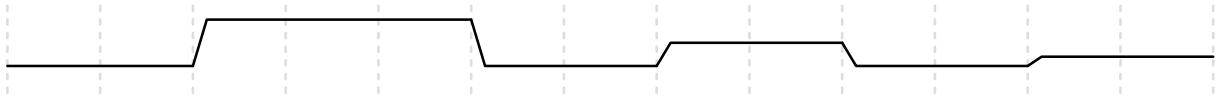
See the *Timing Diagrams* Gallery for more examples.

Variable Voltage Levels

Another Schemdraw extension[Page 96, 1] adds adjustable voltage levels within a signal using the *level* parameter. The value can take 10 different values, specified as digits in the *level* string, where a *1* corresponds to 10%, *2* to 20%, etc., with *0* meaning 100% of the normal high voltage level. As with the *wave* parameter, a period is used to repeat the previous level value. The level parameter only applies to wave type *1*.

Here, the first pulse is 100%, the second at 50%, and the third at 20%.

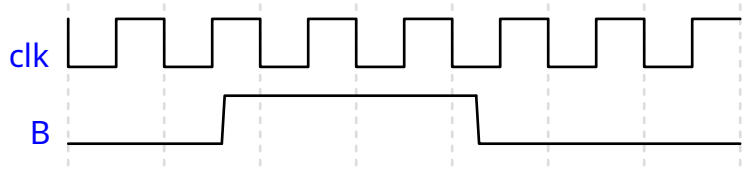
```
logic.TimingDiagram(
    {'signal': [
        {'wave':  '0.1..0.1.0.1.',
         'level': '0......5...2.',
        }],
    })
```



Asynchronous Signals

WaveDrom does not have a means for defining asynchronous signals - all waves must transition on period boundaries. Schemdraw adds asyncrhonous signals using the *async* parameter, as a list of period multiples for each transition in the wave. Note the beginning and end time of the wave must also be specified, so the length of the *async* list must be one more than the length of *wave*.

```
logic.TimingDiagram(
    {'signal': [
        {'name': 'clk', 'wave': 'n......'},
        {'name': 'B', 'wave': '010', 'async': [0, 1.6, 4.25, 7]}]},
    risetime=.03)
```



Shading

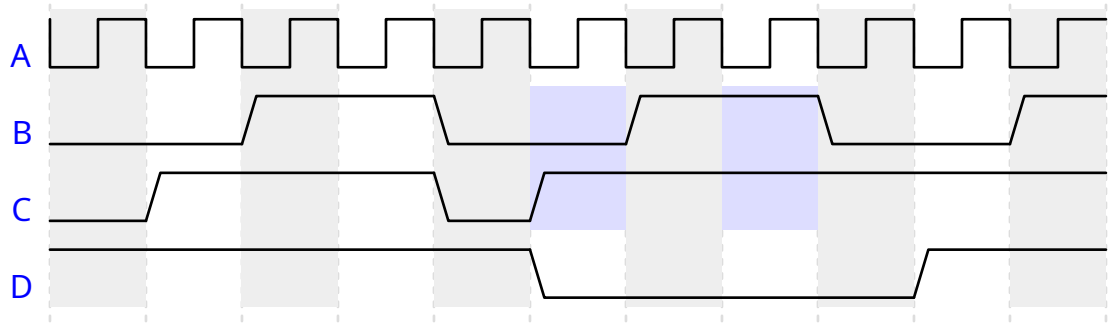
Shading may be added to phases using the *shade* key in the top-level dictionary[Page 96, 1]. Each item in the shade list is a string with the format

```
'Phases Signals Color'
```

The *Phases* parameter may be ‘even’ or ‘odd’, to shade every other phase, or it may be a comma-separated list of phase numbers, such as ‘1,3’ to shade the first and third columns (starting at 0). The *Signals* parameter may be an asterisk *

to shade every signal row, or it may be the first and last rows separated by a colon, for example "1:3" shades the first through third signal rows.

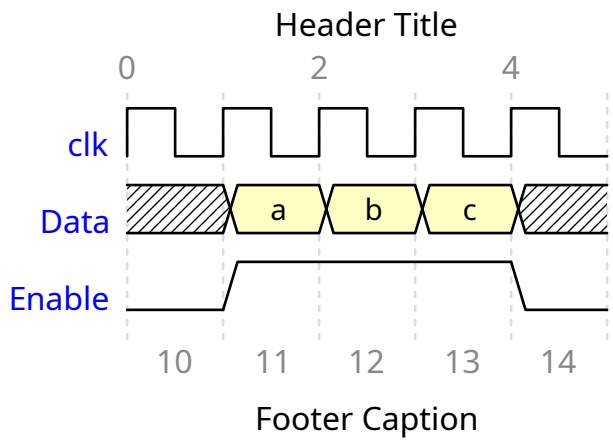
```
logic.TimingDiagram(
    {"signal": [
        {"name": "A", "wave": "n........."},
        {"name": "B", "wave": "0.1.0.1.0.1"},
        {"name": "C", "wave": "01..01....."},
        {"name": "D", "wave": "1....0...1."},
    ],
    "shade": [
        "even * #eee",
        "5,7 1:2 #ddf",
    ]
    })
```



Titles

The *head* and *foot* dictionaries define items to display above and below the diagram, respectively. Titles or captions are added using the *text* key. Phases may be numbered using either *tick*, to number phases at the clock edge, or *tock* to number phases at their center, providing the starting value of the first phase. The *every* key defines which phases to number.

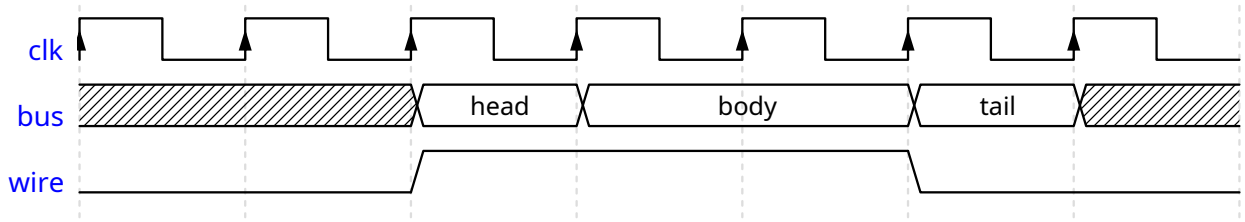
```
logic.TimingDiagram({
    'signal': [
        {'name': 'clk',    'wave': 'p....' },
        {'name': 'Data',   'wave': 'x333x', 'data': 'a b c' },
        {'name':' Enable', 'wave': '01..0' }
    ],
    'head': {
        'text': 'Header Title',
        'tick': 0,
        'every': 2
    },
    'foot': {
        'text': 'Footer Caption',
        'tock': 10
    },
})
```



Configuration

The *config* key provides a dictionary with *hscale*, which may be used to change the width of one period in the diagram:

```
logic.TimingDiagram(
    {'signal': [
        {'name': 'clk', 'wave': 'P......'},
        {'name': 'bus', 'wave': 'x.==.=x', 'data': ['head', 'body', 'tail']},
        {'name': 'wire', 'wave': '0.1..0.'}],
     'config': {'hscale': 2}})
```



Other diagram-level configuration options are specified directly as keyword arguments[Page 96, 1]. These may be overridden by values provided on specific signals.

- **yheight**: Height of one waveform
- **ygap**: Separation between two waveforms
- **risetime**: Rise/fall time for wave transitions
- **fontsize**: Size of label fonts
- **datafontsize**: Size of data font
- **nodesize**: Size of node labels
- **namecolor**: Color for wave names
- **datacolor**: Color for wave data text
- **nodecolor**: Color for node text
- **gridcolor**: Color of background grid
- **edgecolor**: Color of edge notations (default blue)

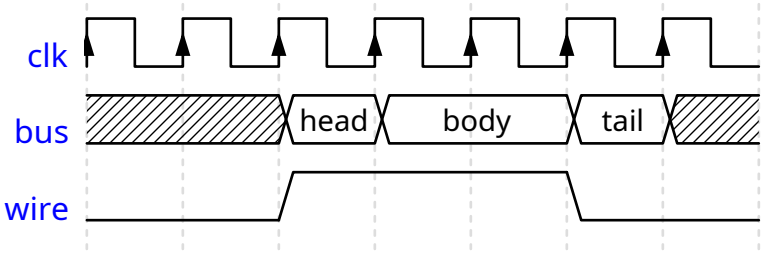
- **tickcolor**: Color of tick/tock labels in head/foot
- **grid**: Enable grid lines (default True)

Using JSON

Because the examples from WaveDrom use JavaScript and JSON, they sometimes cannot be directly pasted into Python as dictionaries. The `schemdraw.logic.timing.TimingDiagram.from_json()` method allows input of the Wave-JSON as a string pasted directly from the Javascript/JSON examples without modification.

Notice lack of quoting on the dictionary keys, requiring the *from_json* method to parse the string.

```
logic.TimingDiagram.from_json('''{ signal: [
  { name: "clk",  wave: "P......" },
  { name: "bus",  wave: "x.==.=x", data: ["head", "body", "tail", "data"] },
  { name: "wire", wave: "0.1..0." }
]}''')
```



Note

TimingDiagram is an *Element*, meaning it may be added to a schemdraw *Drawing* with other schematic components. To save a standalone *TimingDiagram* to an image file, first add it to a drawing, and save the drawing:

```
with schemdraw.Drawing(file='timing.svg'):
    logic.TimingDiagram(
        {'signal': [
            {'name': 'A', 'wave': '0..1..01.'},
            {'name': 'B', 'wave': '101..0...'}]})
```

3.6.2 Bit Field Diagrams

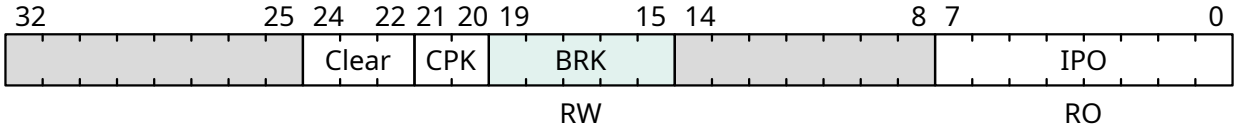
Bit Field diagrams may be drawn using *schemdraw.logic.bitfield.BitField*. They work similar to timing diagrams, with a single parameter dictionary defining the element, which may also be supplied in a *from_json* class method.

```
logic.BitField(
    {'reg': [
        { "name": "IPO",   "bits": 8, "attr": "RO" },
        {                  "bits": 7 },
        { "name": "BRK",   "bits": 5, "attr": "RW", "type": 4 },
        { "name": "CPK",   "bits": 2 },
        { "name": "Clear", "bits": 3 },
```

(continues on next page)

(continued from previous page)

```
        { "bits": 8 }
    ],
    }
)
```



The dictionary passed to BitField may have two keys: *reg* and *config*. The *reg* key is a list of bit groups within the register. Each item in the list may have attributes:

- **name**: Text to display within the bit group
- **bits**: Number of bits within the group
- **attr**: Label to show below the group. May be a string, or integer. If integer, the binary representation is shown. May also be a list of multiple lines.
- **type**: 0-9 color code to fill the bit group. Or may be any valid color string.

Schemdraw adds these parameters not available in the original WaveDrom:

- **scale**: Scale factor for bit width in the group
- **number**: Show first and last bit numbers above the group

The *config* dictionary may include these key-value pairs:

- **lanes**: Number of lanes
- **hflip**: Reverse order of lanes
- **vflip**: Reverse order of bits
- **compact**: Remove whitespace between lanes
- **bits**: Total number of bits to include (padded out if not included in the *reg* list)
- **label**: Dictionary of either 'left' or 'right' and text to display left or right of the lanes.

Additional parameters may be passed directly to *BitField*. Values in the config dictionary above take precedence.

- **bitheight**: Height of a bit register box in drawing units
- **width**: Full width of the register box in drawing units
- **fontsize**: Size of all text labels
- **lw**: Line width for borders
- **ygap**: Distance between lanes. Omit to auto-space based on label heights
- **vflip**: Flip order of bits
- **hflip**: Flip order of lanes
- **compact**: Remove whitespace between lanes

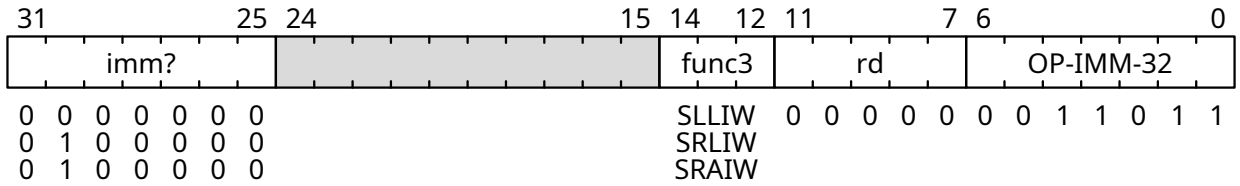
Schemdraw's implementation has these known differences compared to WaveDrom:

- 'type' parameter, which is used to specify a fill color, can be the 0-9 code as in WaveDrom, or any valid color string
- hspace defines the full width of the register in pixels, without including any labels

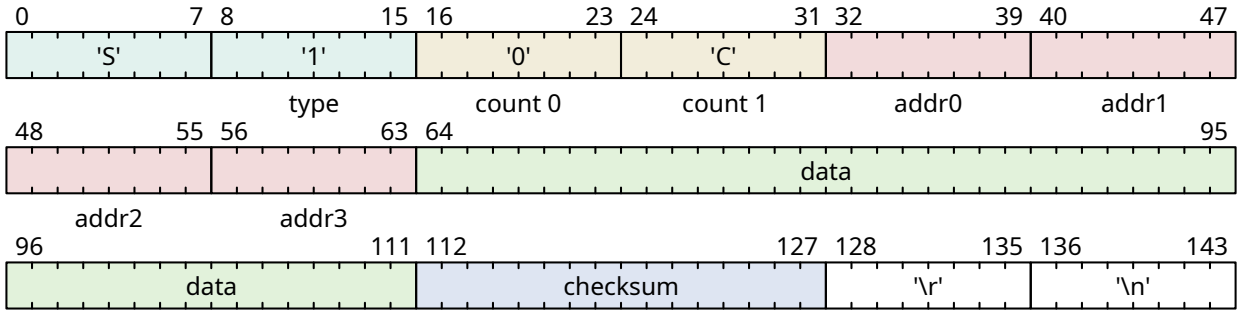
- vspace defines the full width of a register in pixels, without including any labels or padding
- margins are ignored (but can be set by adding the BitField to a schemdraw Drawing)

Examples are below, many borrowed from here.

```
logic.BitField.from_json(r'''
{reg:[
    {name: 'OP-IMM-32', bits: 7,  attr: 0b0011011},
    {name: 'rd',     bits: 5,  attr: 0},
    {name: 'func3',  bits: 3,  attr: ['SLLIW', 'SRLIW', 'SRAIW']},
    {bits: 10},
    {name: 'imm?',   bits: 7, attr: [0, 32, 32]}
]}
'''
)
```




```
logic.BitField.from_json(r'''
{reg: [
{bits: 8, name: "'S'", type: 4},
{bits: 8, name: "'1'", type: 4, attr: 'type'},
{bits: 8, name: "'0'", attr: 'count 0', type: 5},
{bits: 8, name: "'C'", attr: 'count 1', type: 5},
{bits: 8, attr: 'addr0', type: 2},
{bits: 8, attr: 'addr1', type: 2},
{bits: 8, attr: 'addr2', type: 2},
{bits: 8, attr: 'addr3', type: 2},
{bits: 48, name: 'data', type: 6},
{bits: 16, name: 'checksum', type: 7},
{bits: 8, name: "'\\r'"},
{bits: 8, name: "'\\n'"},
], config: {hspace: 800, lanes: 3, bits: 144, vflip: true, hflip: true}}
'''
)
```

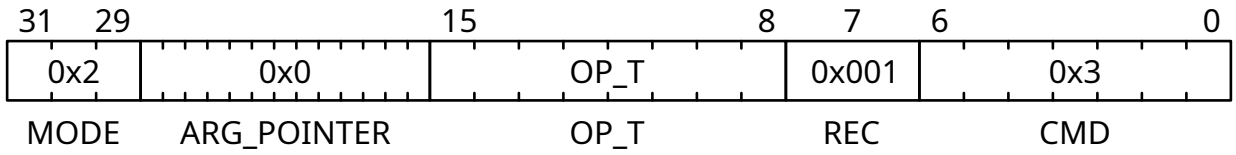



```
logic.BitField.from_json('''
```

(continues on next page)

(continued from previous page)

```
{'reg': [
    {'name': '0x3',   'bits': 7,  attr: 'CMD' },
    {'name': '0x001', 'bits': 1,  attr: 'REC', scale: 3 },
    {'name': 'OP_T',  'bits': 8,  attr: 'OP_T' },
    {'name': '0x0',   'bits': 13, attr: 'ARG_POINTER', scale: .5, number: false },
    {'name': '0x2',   'bits': 3,  attr: 'MODE' },
],
}
''')
```



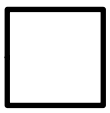
3.7 Signal Processing

Signal processing elements can be drawn by importing the *schemdraw.dsp.dsp* module:

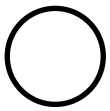
```
from schemdraw import dsp
```

Because each element may have multiple connections in and out, these elements are not 2-terminal elements that extend "leads", so they must be manually connected with *Line* or *Arrow* elements. The square elements define anchors 'N', 'S', 'E', and 'W' for the four directions. Circle-based elements also includ 'NE', 'NW', 'SE', and 'SW' anchors. Directional elements, such as *Amp*, *Adc*, and *Dac* define anchors *input* and *out*.

Square

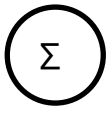


Circle



Sum

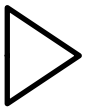
SumSigma



Mixer

Speaker

Amp



OscillatorBox

Oscillator

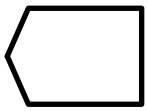
Filter

Filter(response='lp')

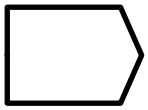
Filter(response='bp')

Filter(response='hp')

Adc



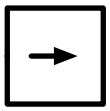
Dac



Demod

Circulator

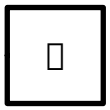
Isolator



VGA

Labels are placed in the center of the element. The generic *Square* and *Circle* elements can be used with a label to define other operations. For example, an integrator may be created using:

```
dsp.Square().label(r'$\int$')
```



3.8 Image-based Elements

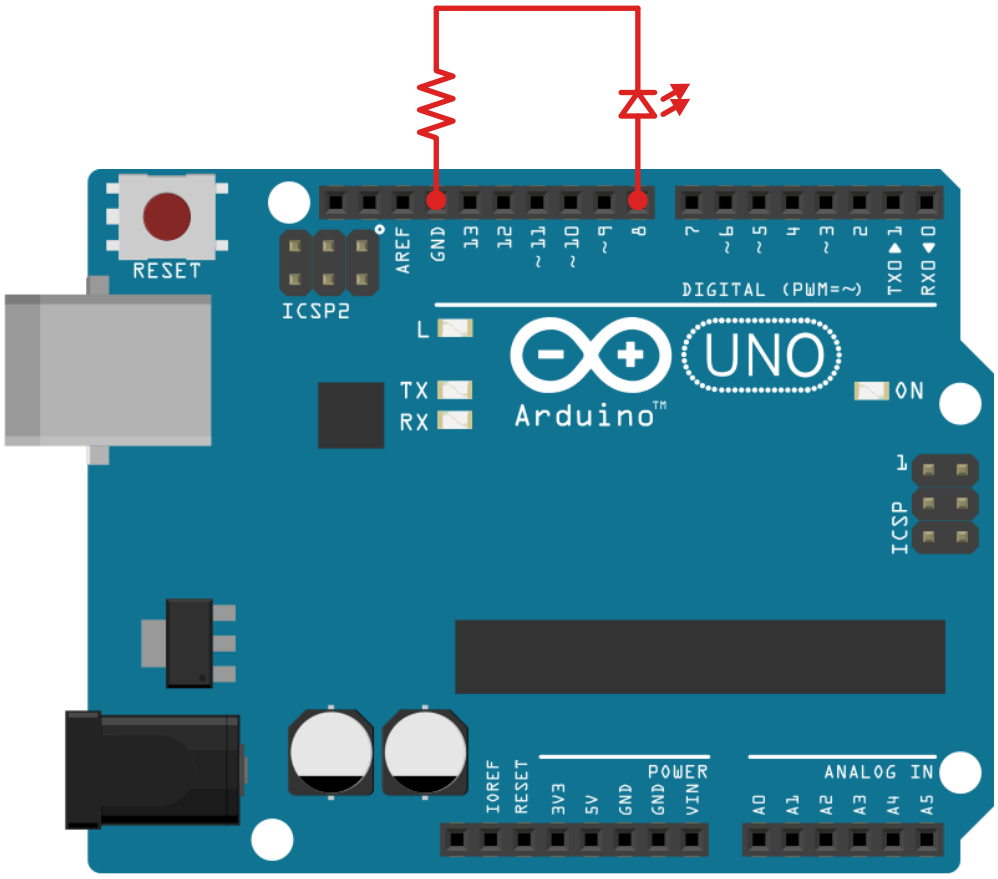
Elements can be made from image files, including PNG and SVG images, using *schemdraw.elements.ElementImage*. SVG images are only supported in the SVG backend at this time (See *Backends*).

Common usage is to subclass *ElementImage* and provide the image file and anchor positions in the subclass's init method. For example, an Arduino Board element may be created from an image:

```
class ArduinoUno(elm.ElementImage):
    ''' Arduino Element '''
    def __init__(self):
        # Dimensions based on image size to make it about the right
        # size relative to other components
        width = 10.3
        height = width/1.397
        pinspacing = .35   # Spacing between header pins

        super().__init__('ArduinoUNO.png', width=width, height=height, xy=(-.75, 0))

        # Only defining the top header pins as anchors for now
        top = height * .956
        arefx = 3.4
        for i, pinname in enumerate(['aref', 'gnd', 'pin13', 'pin12', 'pin11',
                                     'pin10', 'pin9', 'pin8']):
            self.anchors[pinname] = (arefx + i*pinspacing, top)
```

The Arduino element is used like any other element:

```
with schemdraw.Drawing() as d:
    d.config(color='#dd2222', unit=2)
    arduino = ArduinoUno()
    elm.Dot().at(arduino.gnd)
    elm.Resistor().up().scale(.7)
    elm.Line().right().tox(arduino.pin8)
    elm.LED().down().reverse().toy(arduino.pin8).scale(.7)
    elm.Dot().at(arduino.pin8)
```



Arduino Image Source , CC-BY-SA-3.0.

See *Pictorial Elements* for using Image Elements with other graphical schematic components.

3.9 Pictorial Elements

Pictorial Schematics use pictures, rather than symbols, to represent circuit elements. Schemdraw provides a few common pictorial elements, and more may be added by loading in *Image-based Elements*.

All the built-in pictorial elements are drawn below.

```
from schemdraw import pictorial
```

CapacitorCeramic

CapacitorMylar

CapacitorElectrolytic

TO92

LED

LEDOrange

LEDYellow

LEDGreen

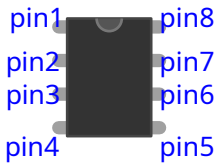
LEDBlue

LEDWhite

Diode

Resistor

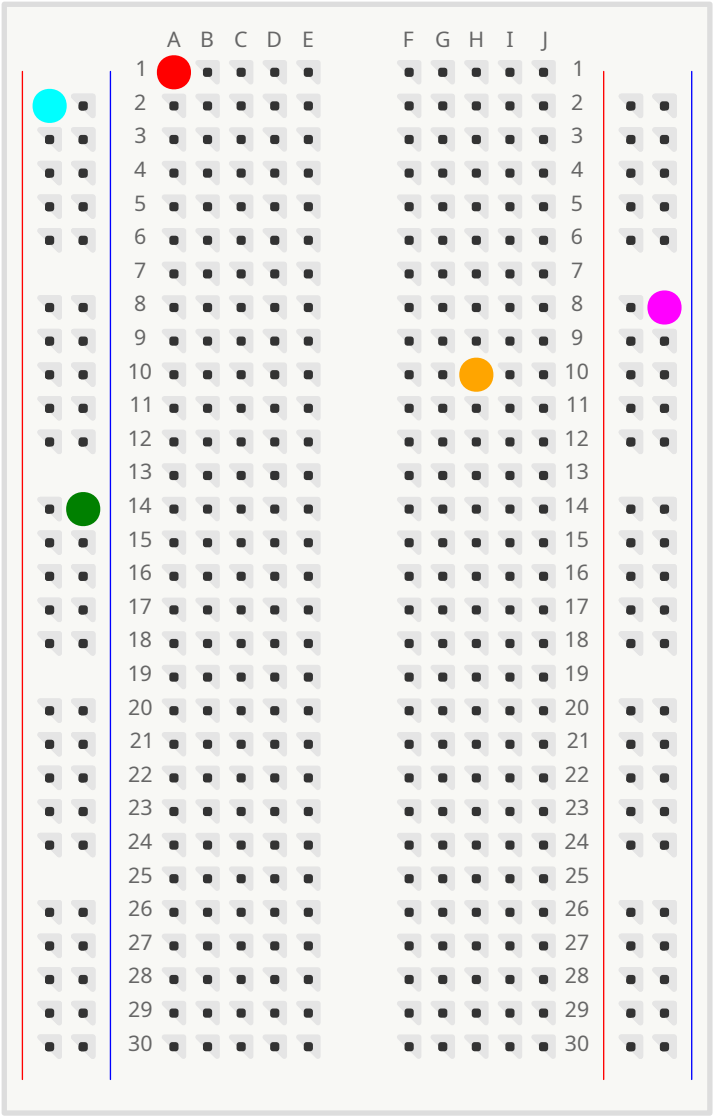
DIP



3.9.1 Breadboard

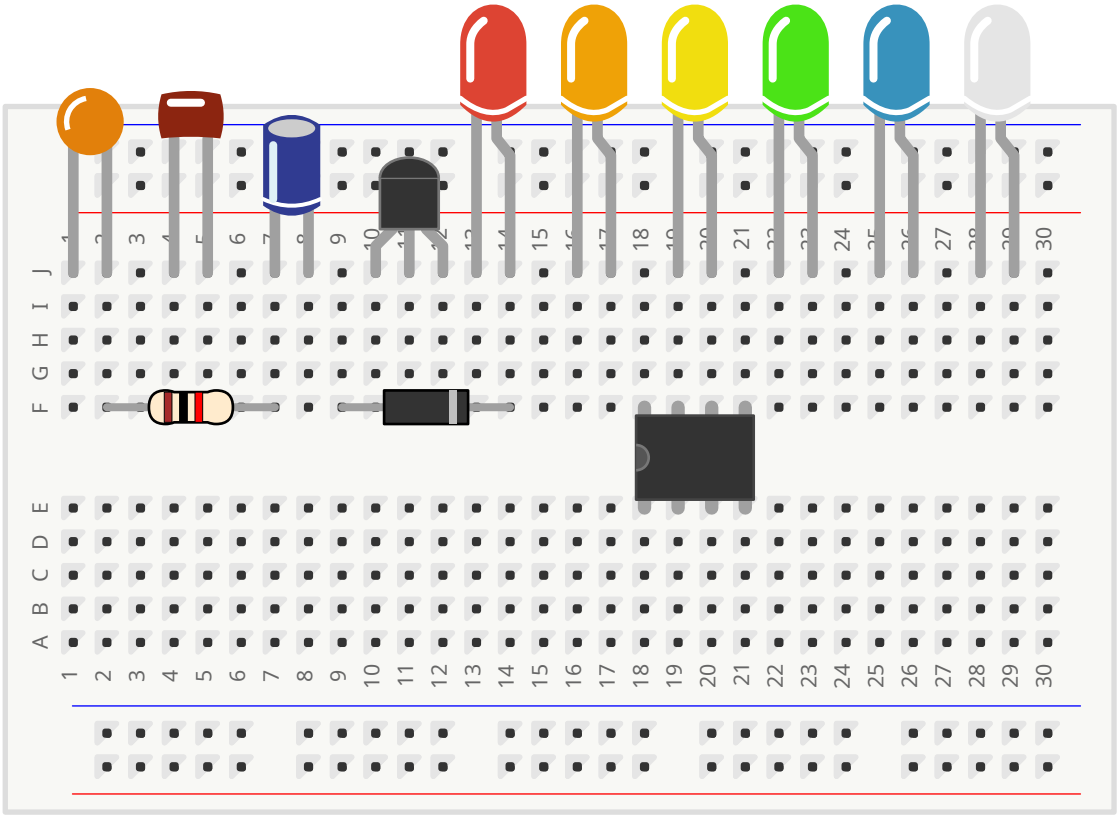
The *schemdraw.pictorial.pictorial.Breadboard* element has anchors at each pin location. Anchor names for the center block of pins use the column letter and row number, such as *A1* for the top left pin. The power strip columns along each side are *L1_x* for the first column on the left side or *L2_x*, for the second column on the left side, with *x* designating the row number. Similarly, *R1_x* and *R2_x* designate anchors on the right power strip columns. Note the "missing" rows in the power strip columns, such as row 6, are not defined. Some examples are shown below.

```
with schemdraw.Drawing():
    bb = pictorial.Breadboard()
    elm.Dot(radius=.15).at(bb.A1).color('red')
    elm.Dot(radius=.15).at(bb.H10).color('orange')
    elm.Dot(radius=.15).at(bb.L1_1).color('cyan')
    elm.Dot(radius=.15).at(bb.R2_7).color('magenta')
    elm.Dot(radius=.15).at(bb.L2_13).color('green')
```



The following example shows the all elements drawn on a breadboard.

```
with schemdraw.Drawing():
    bb = pictorial.Breadboard().up()
    pictorial.CapacitorCeramic().at(bb.J1)
    pictorial.CapacitorMylar().at(bb.J4)
    pictorial.CapacitorElectrolytic().at(bb.J7)
    pictorial.TO92().at(bb.J10)
    pictorial.LED().at(bb.J13)
    pictorial.LEDOrange().at(bb.J16)
    pictorial.LEDYellow().at(bb.J19)
    pictorial.LEDGreen().at(bb.J22)
    pictorial.LEDBlue().at(bb.J25)
    pictorial.LEDWhite().at(bb.J28)
    pictorial.Diode().at(bb.F9).to(bb.F14)
    pictorial.Resistor().at(bb.F2).to(bb.F7)
    pictorial.DIP().at(bb.E18).up()
```



3.9.2 Resistors

Resistors and Diodes inherit from *schemdraw.elements.Element2Term*, meaning they may be extended to any length. Resistors take *value* and *tolerance* arguments used to set the color bands. The colors will be the closest possible color code using 3 bands to represent the value.

```
with schemdraw.Drawing():
    pictorial.Resistor(100)
    pictorial.Resistor(220)
    pictorial.Resistor(520)
    pictorial.Resistor(10000)
```

3.9.3 Dual-inline Packages (DIP)

Integrated circuits in DIP packages may be drawn with the *schemdraw.pictorial.pictorial.DIP* element. The *npins* argument sets the total number of pins and *wide* argument specifies a wide-body (0.6 inch) versus the narrow-body (0.3 inch) package.

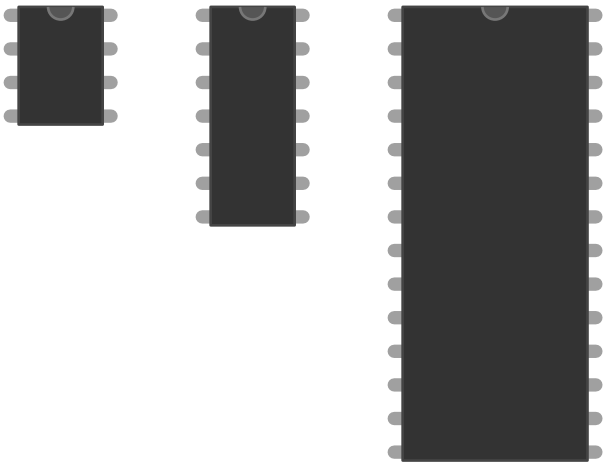
DIPs have anchors *pinX*, where *X* is the pin number.

```
with schemdraw.Drawing():
    pictorial.DIP()
```

(continues on next page)

(continued from previous page)

```
    pictorial.DIP(npins=14).at((2, 0))
    pictorial.DIP(npins=28, wide=True).at((4, 0))
```



3.9.4 Colors

The pictorial elements are drawn using solid shapes. As such, the *.fill()* method must be used to change their color, while the *.color()* method will set only the color of the outline, if the Element has one. For example, to create a custom-color LED:

```
pictorial.LED().fill('purple')
```

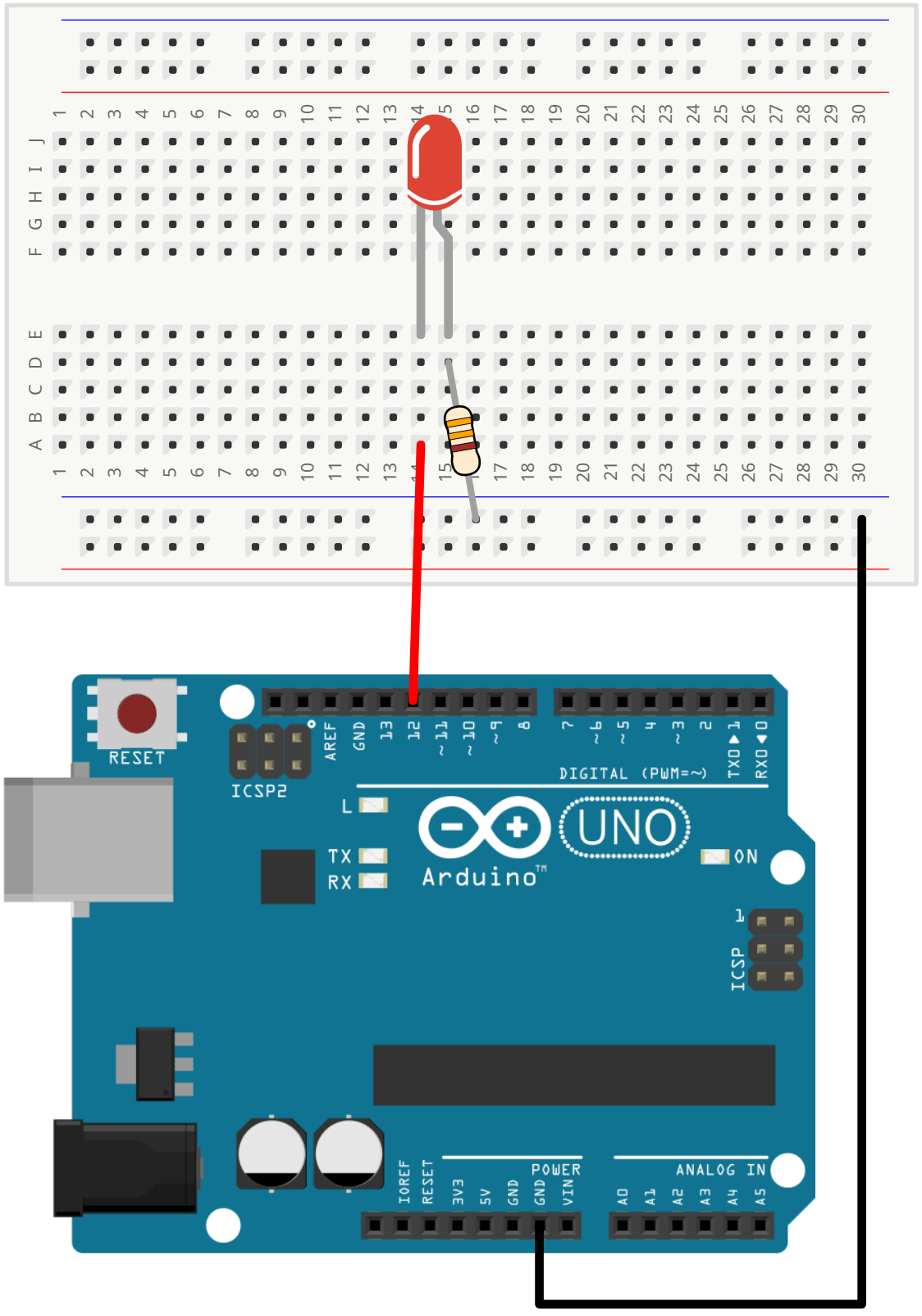
3.9.5 Dimensions

The pictorial elements are designed with spacing so they fit together in a breadboard with 0.1 inch spacing between pins. Some constants are defined to assist in creating other pictorial elements: *pictorial.INCH* and *pictorial.MILLIMETER* convert inches and millimeters to schemdraw's drawing units. *pictorial.PINSPACING* is equal to 0.1 inch, the standard spacing between breadboard and DIP pins.

3.9.6 Example

This example combines an *schemdraw.elements.ElementImage* of an Arduino Uno board with pictorial elements.

```
class ArduinoUno(elm.ElementImage):
    ''' Arduino Element '''
    def __init__(self):
        width = 10.3  # Set the width to scale properly for 0.1 inch pin spacing on␣
↪headers
        height = width/1.397  # Based on image dimensions
        super().__init__('ArduinoUNO.png', width=width, height=height, xy=(-.75, 0))

        # Define all the anchors
        top = height * .956
        arefx = 3.4
        pinspace = pictorial.PINSPACING
        for i, pinname in enumerate(['aref', 'gnd_top', 'pin13', 'pin12', 'pin11',
                                     'pin10', 'pin9', 'pin8']):
            self.anchors[pinname] = (arefx + i*pinspace, top)

        bot = .11*pictorial.INCH
        botx = 1.23*pictorial.INCH
        for i, pinname in enumerate(['ioref', 'reset', 'threev3',
                                     'fivev', 'gnd1', 'gnd2', 'vin']):
            self.anchors[pinname] = (botx + i*pinspace, bot)

        botx += i*pinspace + pictorial.PINSPACING*2
        for i, pinname in enumerate(['A0', 'A1', 'A2', 'A3', 'A4', 'A5']):
            self.anchors[pinname] = (botx + i*pinspace, bot)


with schemdraw.Drawing():
    ard = ArduinoUno()
    bb = pictorial.Breadboard().at((0, 9)).up()
    elm.Wire('n', k=-1).at(ard.gnd2).to(bb.L2_29).linewidth(4)
    elm.Wire().at(ard.pin12).to(bb.A14).color('red').linewidth(4)
    pictorial.LED().at(bb.E14)
    pictorial.Resistor(330).at(bb.D15).to(bb.L2_15)
```



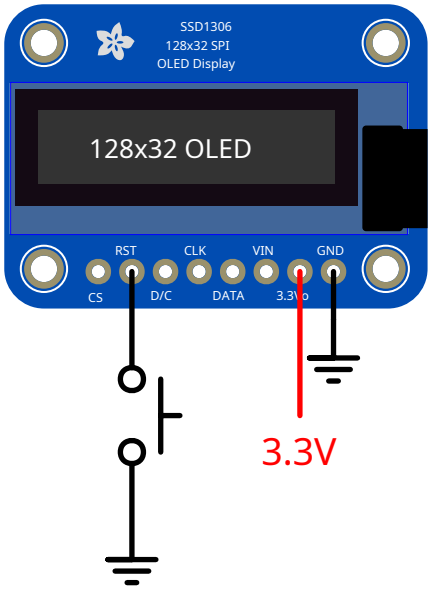
Arduino Image Source , CC-BY-SA-3.0.

3.9.7 Fritzing Part Files

Schemdraw can import part files in the Fritzing format and use them in pictorial schematics. Use *schemdraw.pictorial.fritz.FritzingPart* and provide the file name of an *.fzpz* or *.fzbz* part file. Schemdraw's anchors will be set based on the part "connectors" defined in the part file. In this example, a part is downloaded from the Adafruit Fritzing Library and used in a drawing.

Because Fritzing images are SVG format, *FritzingPart* only works in schemdraw's SVG backend (*Backends*).

```
schemdraw.use('svg')
from urllib.request import urlretrieve
part = 'https://github.com/adafruit/Fritzing-Library/raw/master/parts/Adafruit%20OLED
↪%20Monochrome%20128x32%20SPI.fzpz'
fname, msg = urlretrieve(part)

with schemdraw.Drawing() as d:
    oled = pictorial.FritzingPart(fname)
    elm.Line().down().at(oled.GND).length(.5)
    elm.Ground()
    elm.Line().down().at(oled['3.3V']).color('red').length(1.5).label('3.3V', loc='left')
    elm.Button().at(oled.RESET)
    elm.Ground(lead=False)
```



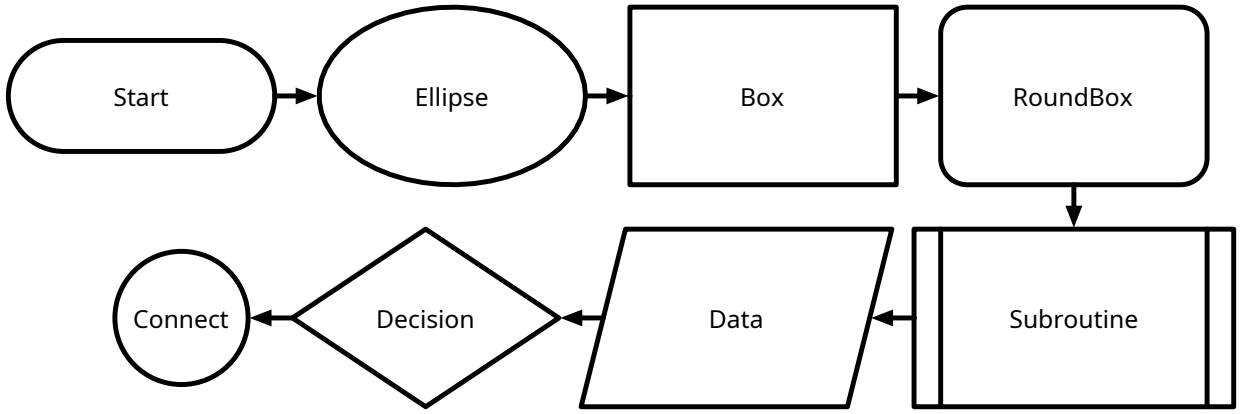
Note that occasionally anchor names defined in Fritzing parts are not valid as Python identifiers, such as the *3.3V* anchor above, and therefore cannot be used as attributes of the element instance (*oled.3.3V* doesn't work, obviously). In these cases, the anchor must be accessed through getitem attribute *oled['3.3V']*.

3.10 Flowcharts and Diagrams

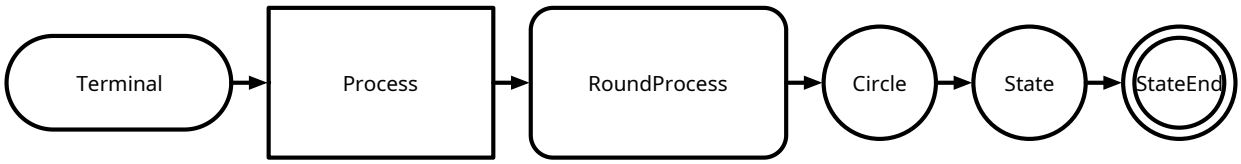
Schemdraw provides basic symbols for flowcharting and state diagrams. The *schemdraw.flow.flow* module contains a set of functions for defining flowchart blocks and connecting lines that can be added to schemdraw Drawings.

```
from schemdraw import flow
```

Flowchart blocks:



Some elements have been defined with multiple names, which can be used depending on the context or user preference:



All flowchart symbols have 16 anchor positions named for the compass directions: 'N', 'S', 'E', 'W', 'NE', 'SE, 'NNE', etc., plus a 'center' anchor.

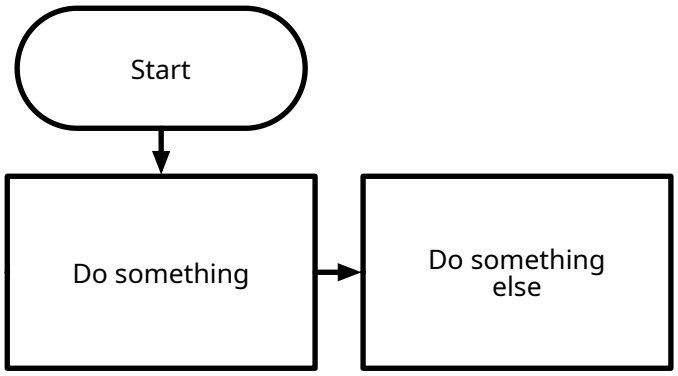
The *schemdraw.elements.intcircuits.Ic* element can be used with the flowchart elements to create blocks with other inputs/outputs per side if needed.

The size of each block must be specified manually using *w* and *h* or *r* parameters to size each block to fit any labels.

3.10.1 Connecting Lines

Typical flowcharts will use *Line* or *Arrow* elements to connect the boxes. The line and arrow elements have been included in the *flow* module for convenience.

```
with schemdraw.Drawing() as d:
    d.config(fontsize=10, unit=.5)
    flow.Terminal().label('Start')
    flow.Arrow()
    flow.Process().label('Do something').drop('E')
    flow.Arrow().right()
    flow.Process().label('Do something\nelse')
```

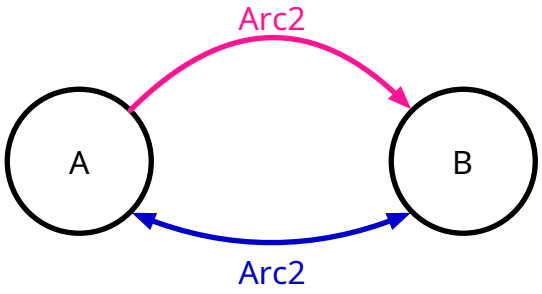


Some flow diagrams, such as State Machine diagrams, often use curved connectors between states. Several Arc connectors are available. Each Arc element takes an *arrow* parameter, which may be '->', '<-', or '<->', to define the end(s) on which to draw arrowheads.

Arc2

Arc2 draws a symmetric quadratic Bezier curve between the endpoints, with curvature controlled by parameter *k*. Endpoints of the arc should be specified using *at()* and *to()* methods.

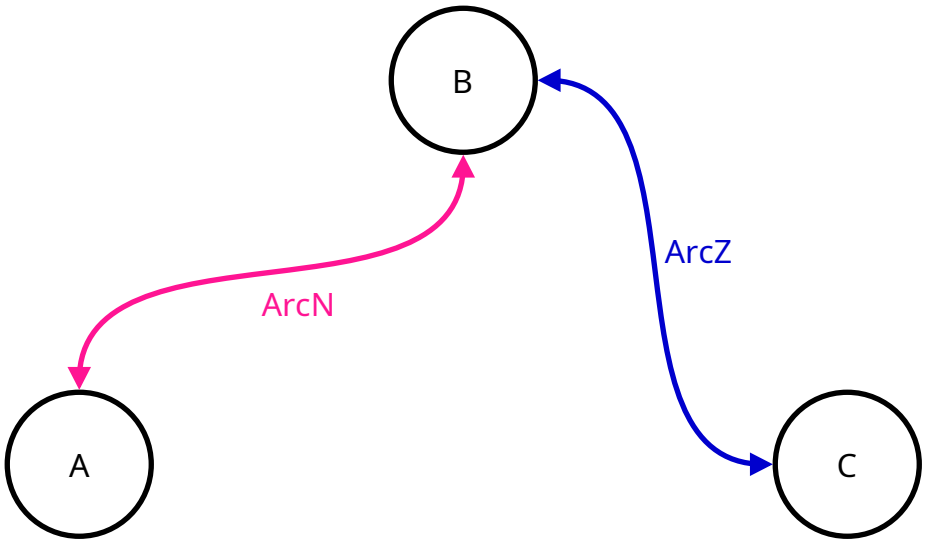
```
with schemdraw.Drawing(fontsize=12, unit=1):
    a = flow.State().label('A')
    b = flow.State(arrow='->').label('B').at((4, 0))
    flow.Arc2(arrow='->').at(a.NE).to(b.NW).color('deeppink').label('Arc2')
    flow.Arc2(k=.2, arrow='<->').at(b.SW).to(a.SE).color('mediumblue').label('Arc2')
```



ArcZ and ArcN

These draw symmetric cubic Bezier curves between the endpoints. The *ArcZ* curve approaches the endpoints horizontally, and *ArcN* approaches them vertically.

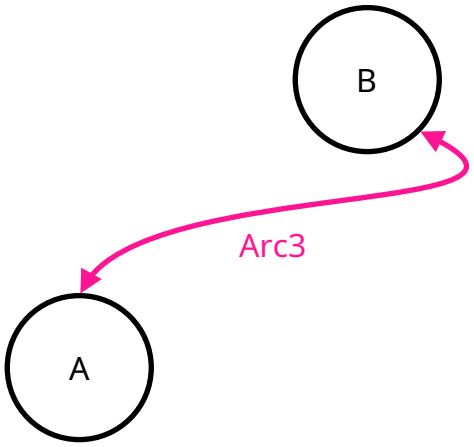
```
with schemdraw.Drawing(fontsize=12, unit=1):
    a = flow.State().label('A')
    b = flow.State().label('B').at((4, 4))
    c = flow.State().label('C').at((8, 0))
    flow.ArcN(arrow='<->').at(a.N).to(b.S).color('deeppink').label('ArcN')
    flow.ArcZ(arrow='<->').at(b.E).to(c.W).color('mediumblue').label('ArcZ')
```



Arc3

The *Arc3* curve is an arbitrary cubic Bezier curve, defined by endpoints and angle of approach to each endpoint. *ArcZ* and *ArcN* are simply *Arc3* defined with the angles as 0 and 180, or 90 and 270, respectively.

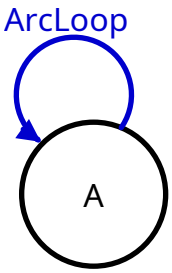
```
with schemdraw.Drawing(fontsize=12, unit=1):
    a = flow.State().label('A')
    b = flow.State().at((3, 3))
    flow.Arc3(th1=75, th2=-45, arrow='<->').at(a.N).to(b.SE).color('deeppink').label(
↪'Arc3')
```



ArcLoop

The *ArcLoop* curve draws a partial circle that intersects the two endpoints, with the given radius. Often used in state machine diagrams to indicate cases where the state does not change.

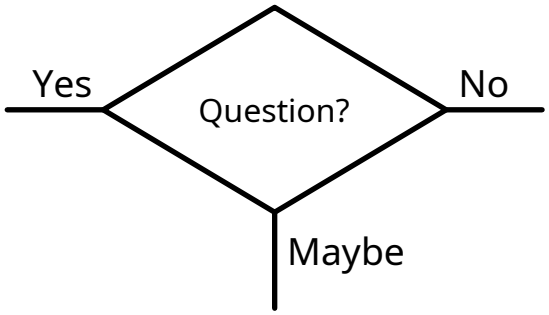
```
with schemdraw.Drawing(fontsize=12, unit=1):
    a = flow.State().label('A')
    flow.ArcLoop(arrow='<-').at(a.NW).to(a.NNE).color('mediumblue').label('ArcLoop',␣
↪halign='center')
```



3.10.2 Decisions

To label the decision branches, the *schemdraw.flow.flow.Decision* element takes keyword arguments for each cardinal direction. For example:

```
decision = flow.Decision(W='Yes', E='No', S='Maybe').label('Question?')
```



3.10.3 Layout and Flow

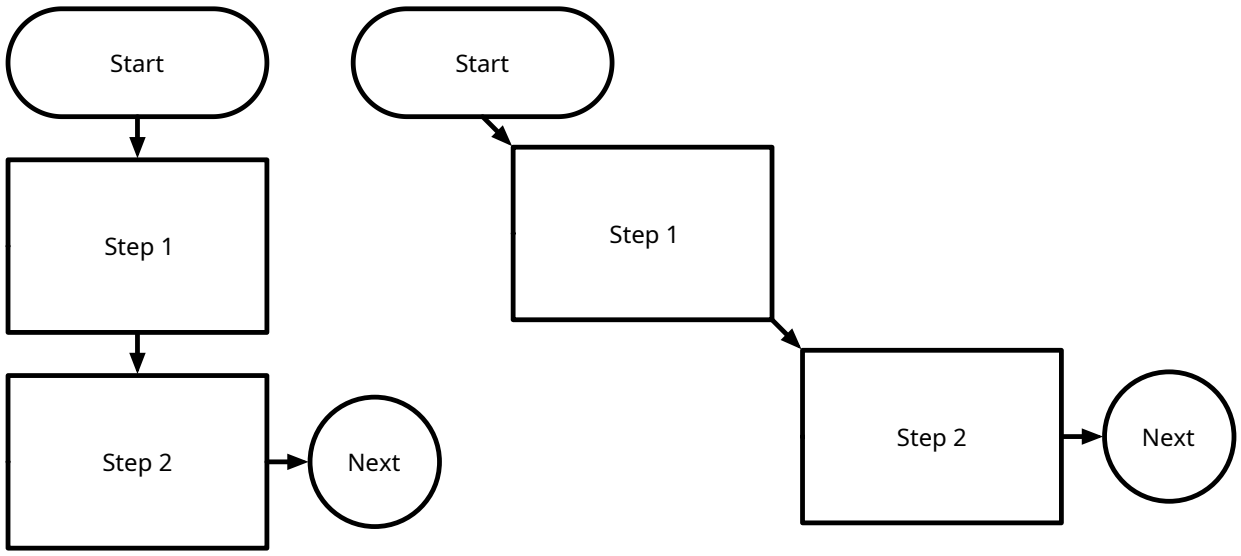
Without any directions specified, boxes flow top to bottom (see left image). If a direction is specified (right image), the flow will continue in that direction, starting the next arrow at an appropriate anchor. Otherwise, the *drop* method is useful for specifing where to begin the next arrow.

```
with schemdraw.Drawing() as d:
    d.config(fontsize=10, unit=.5)
    flow.Terminal().label('Start')
    flow.Arrow()
    flow.Process().label('Step 1')
    flow.Arrow()
    flow.Process().label('Step 2').drop('E')
    flow.Arrow().right()
```

(continues on next page)

(continued from previous page)

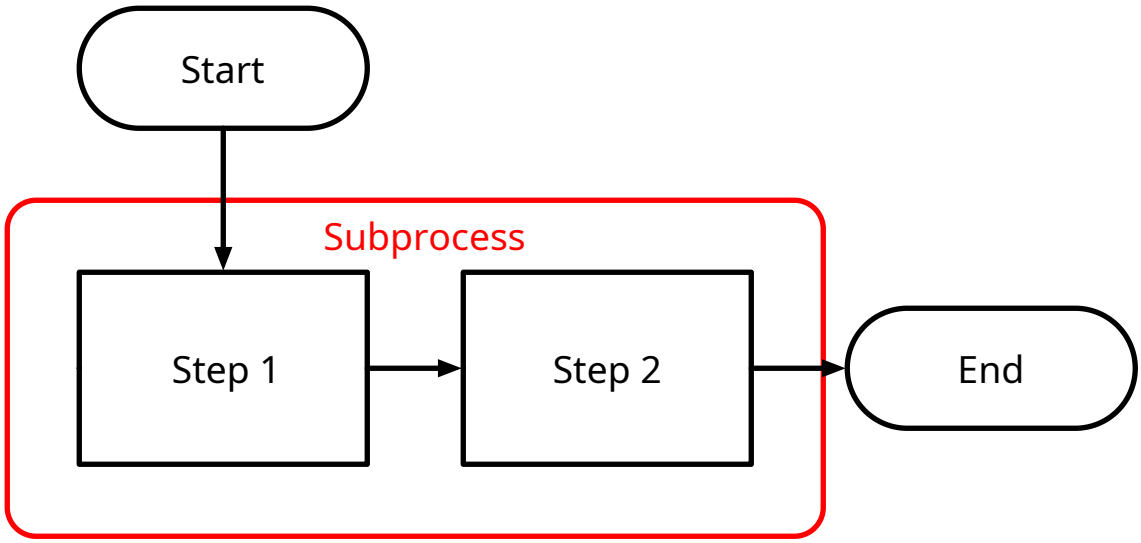
```
    flow.Connect().label('Next')

    flow.Terminal().label('Start').at((4, 0))
    flow.Arrow().theta(-45)
    flow.Process().label('Step 1')
    flow.Arrow()
    flow.Process().label('Step 2').drop('E')
    flow.Arrow().right()
    flow.Connect().label('Next')
```



3.10.4 Containers

Use *schemdraw.Drawing.container()* as a context manager to add elements to be enclosed in a box. The elements in the container are added to the outer drawing too; the *container* just draws the box around them when it exits the *with*.

```
with schemdraw.Drawing(unit=1) as d:
    flow.Start().label('Start')
    flow.Arrow().down(1.5)
    with d.container() as c:
        flow.Box().label('Step 1').drop('E')
        flow.Arrow().right()
        flow.Box().label('Step 2')
        c.color('red')
        c.label('Subprocess', loc='N', halign='center', valign='top')
    flow.Arrow().right()
    flow.Start().label('End').anchor('W')
```



Containers may be nested, calling *container()* on either a Drawing, or another Container.

3.10.5 Examples

See the *Flowcharting* Gallery for more examples.

CHAPTER

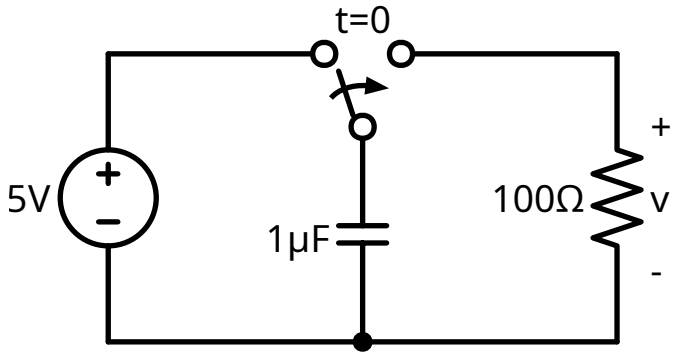
FOUR

CIRCUIT GALLERY

4.1 Analog Circuits

4.1.1 Discharging capacitor

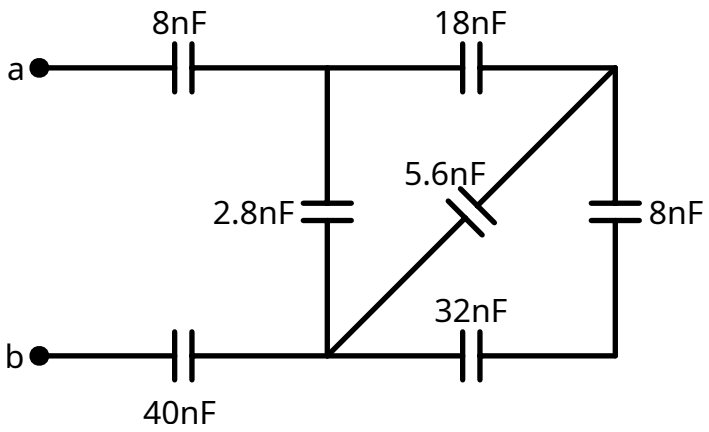
Shows how to connect to a switch with anchors.




```
with schemdraw.Drawing() as d:
    V1 = elm.SourceV().label('5V')
    elm.Line().right(d.unit*.75)
    S1 = elm.SwitchSpdt2(action='close').up().anchor('b').label('$t=0$', loc='rgt')
    elm.Line().right(d.unit*.75).at(S1.c)
    elm.Resistor().down().label(r'$100\Omega$').label(['+','$v_o$','-'], loc='bot')
    elm.Line().to(V1.start)
    elm.Capacitor().at(S1.a).toy(V1.start).label(r'1$\mu$F').dot()
```

4.1.2 Capacitor Network

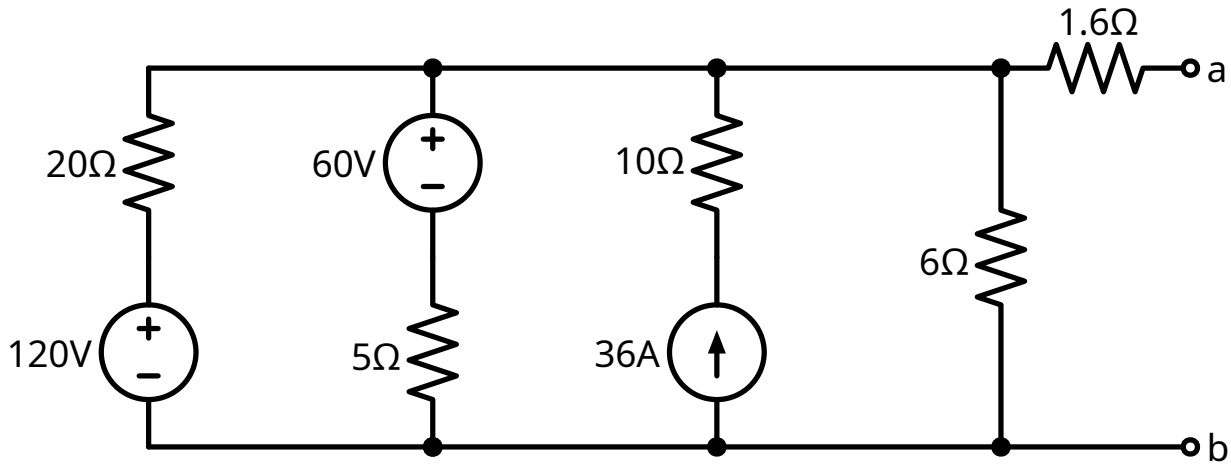
Shows how to use endpoints to specify exact start and end placement.




```
with schemdraw.Drawing() as d:
    d.config(fontsize=12)
    C1 = elm.Capacitor().label('8nF').idot().label('a', 'left')
    C2 = elm.Capacitor().label('18nF')
    C3 = elm.Capacitor().down().label('8nF', loc='bottom')
    C4 = elm.Capacitor().left().label('32nF')
    C5 = elm.Capacitor().label('40nF', loc='bottom').dot().label('b', 'left')
    C6 = elm.Capacitor().endpoints(C1.end, C5.start).label('2.8nF')
    C7 = (elm.Capacitor().endpoints(C2.end, C5.start)
                .label('5.6nF', loc='center', ofst=(-.3, -.1), halign='right', valign=
↪'bottom'))
```

4.1.3 ECE201-Style Circuit

This example demonstrate use of *hold()* and using the *tox()* and *toy()* methods.




```
with schemdraw.Drawing() as d:
    d.config(unit=2)  # unit=2 makes elements have shorter than normal leads
    with d.hold():
        R1 = elm.Resistor().down().label('20Ω')
        V1 = elm.SourceV().down().reverse().label('120V')
```

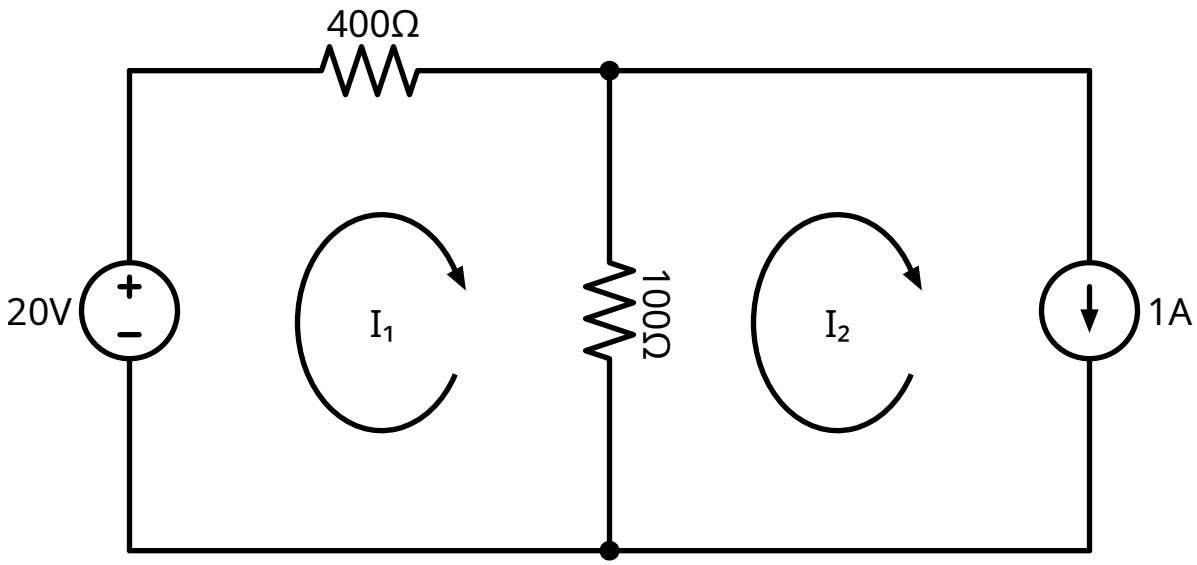
(continues on next page)

(continued from previous page)

```
        elm.Line().right(3).dot()
    elm.Line().right(3).dot()
    elm.SourceV().down().reverse().label('60V')
    elm.Resistor().label('5Ω').dot()
    elm.Line().right(3).dot()
    elm.SourceI().up().label('36A')
    elm.Resistor().label('10Ω').dot()
    elm.Line().left(3).hold()
    elm.Line().right(3).dot()
    R6 = elm.Resistor().toy(V1.end).label('6Ω').dot()
    elm.Line().left(3).hold()
    elm.Resistor().right().at(R6.start).label('1.6Ω').dot(open=True).label('a', 'right')
    elm.Line().right().at(R6.end).dot(open=True).label('b', 'right')
```

4.1.4 Loop Currents

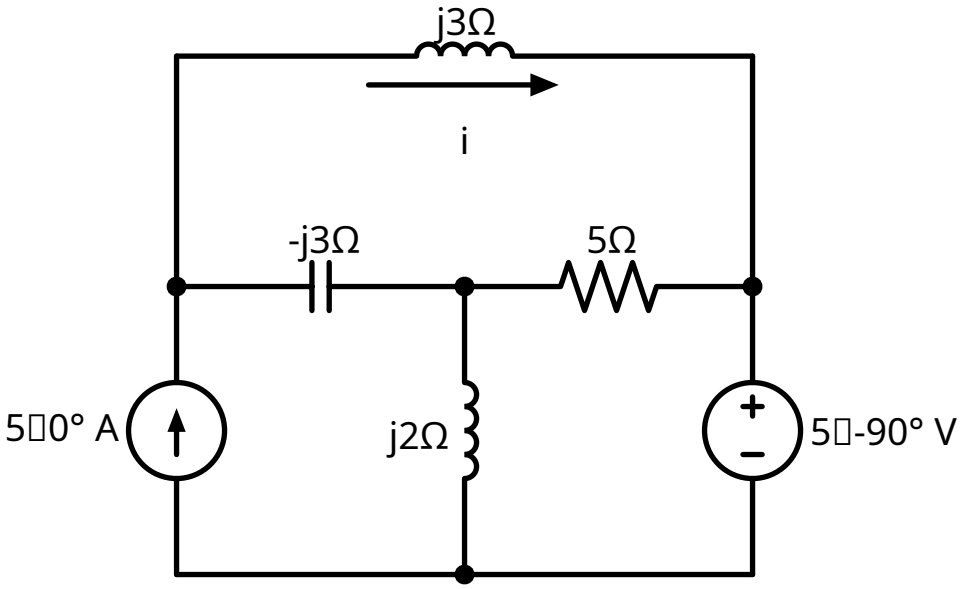
Using the *schemdraw.elements.lines.LoopCurrent* element to add loop currents, and rotating a label to make it fit.




```
with schemdraw.Drawing() as d:
    d.config(unit=5)
    V1 = elm.SourceV().label('20V')
    R1 = elm.Resistor().right().label('400Ω').dot()
    R2 = elm.Resistor().down().label('100Ω', loc='bot', rotate=True).dot().hold()
    L1 = elm.Line().right()
    I1 = elm.SourceI().down().label('1A', loc='bot')
    L2 = elm.Line().tox(V1.start)
    elm.LoopCurrent([R1,R2,L2,V1], pad=1.25).label('$I_1$')
    elm.LoopCurrent([R1,I1,L2,R2], pad=1.25).label('$I_2$')    # Use R1 as top element␣
↪for both so they get the same height
```

4.1.5 AC Loop Analysis

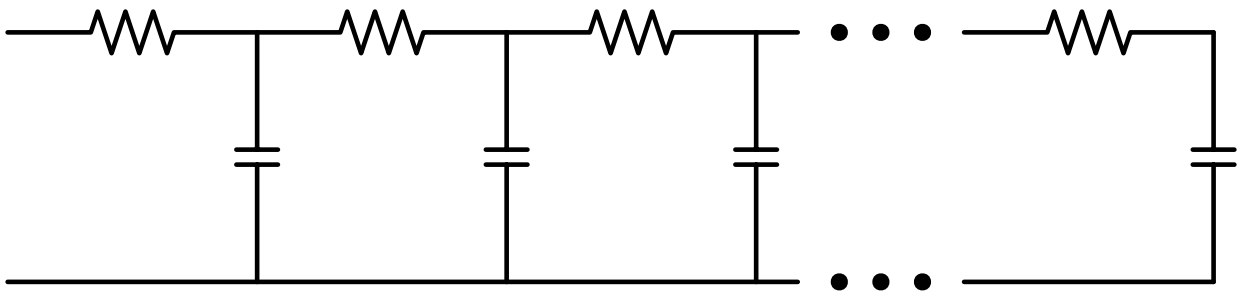
Another good problem for ECE students…




```
with schemdraw.Drawing() as d:
    I1 = elm.SourceI().label('5∠0° A').dot()
    with d.hold():
        elm.Capacitor().right().label('-j3Ω').dot()
        elm.Inductor().down().label('j2Ω').dot().hold()
        elm.Resistor().right().label('5Ω').dot()
        V1 = elm.SourceV().down().reverse().label('5∠-90° V', loc='bot')
        elm.Line().tox(I1.start)
    elm.Line().up(d.unit*.8)
    L1 = elm.Inductor().tox(V1.start).label('j3Ω')
    elm.Line().down(d.unit*.8)
    elm.CurrentLabel(top=False, ofst=.3).at(L1).label('$i_g$')
```

4.1.6 Infinite Transmission Line

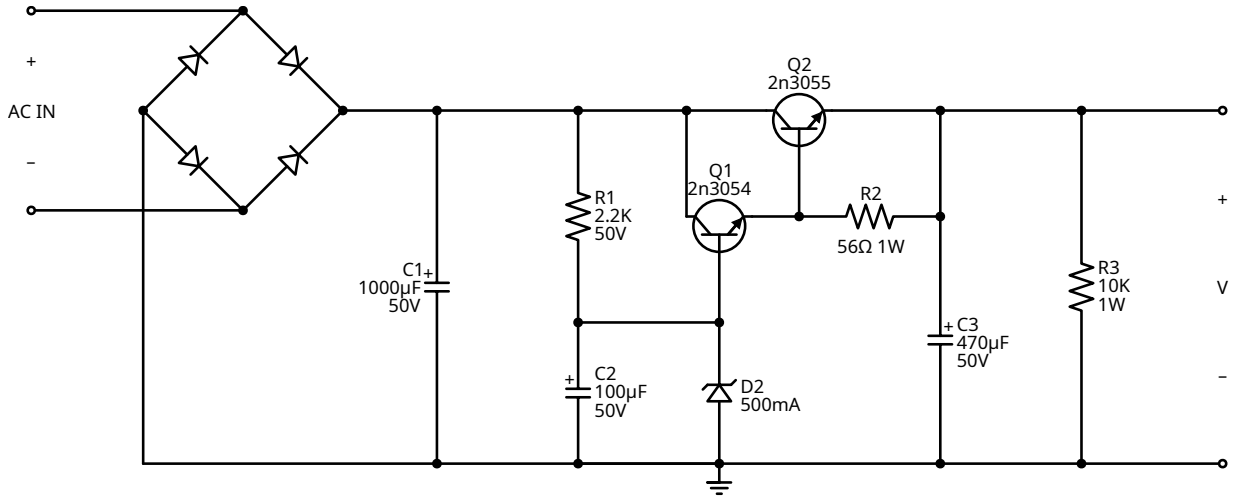
Elements can be added inside for-loops if you need multiples. The ellipsis is just another circuit element, called *DotDotDot* since Ellipsis is a reserved keyword in Python. This also demonstrates the *schemdraw.elements.ElementDrawing* class to merge multiple elements into a single definition.



```
with schemdraw.Drawing(show=False) as d1:
    elm.Resistor()
    with d1.hold():
        elm.Capacitor().down()
        elm.Line().left()

with schemdraw.Drawing() as d2:
    for i in range(3):
        elm.ElementDrawing(d1)

    with d2.hold():
        elm.Line().length(d2.unit/6)
        elm.DotDotDot()
        elm.ElementDrawing(d1)
    d2.move(dy=-d2.unit)
    elm.Line().right(d2.unit/6)
    elm.DotDotDot()
```

4.1.7 Power supply

Notice the diodes could be added individually, but here the built-in *Rectifier* element is used instead. Also note the use of newline characters inside resistor and capacitor labels.




```
with schemdraw.Drawing() as d:
    d.config(inches_per_unit=.5, unit=3)
    D = elm.Rectifier()
    elm.Line().left(d.unit*1.5).at(D.N).dot(open=True).idot()
    elm.Line().left(d.unit*1.5).at(D.S).dot(open=True).idot()
    G = elm.Gap().toy(D.N).label(['-', 'AC IN', '+'])

    top = elm.Line().right(d.unit*3).at(D.E).idot()
    Q2 = elm.BjtNpn(circle=True).up().anchor('collector').label('Q2\n2n3055')
    elm.Line().down(d.unit/2).at(Q2.base)
    Q2b = elm.Dot()
    elm.Line().left(d.unit/3)
```

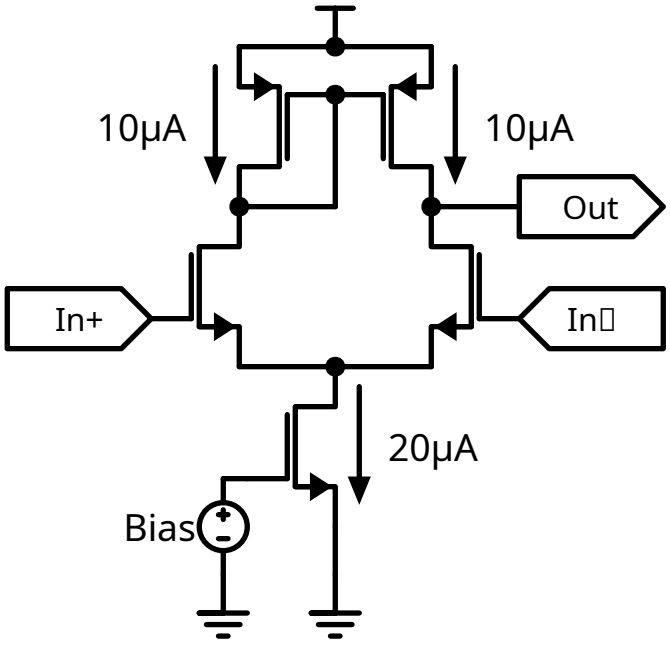
(continues on next page)

(continued from previous page)

```
    Q1 = elm.BjtNpn(circle=True).up().anchor('emitter').label('Q1\n    2n3054')
    elm.Line().at(Q1.collector).toy(top.center).dot()

    elm.Line().down(d.unit/2).at(Q1.base).dot()
    elm.Zener().down().reverse().label('D2\n500mA', loc='bot').dot()
    G = elm.Ground()
    elm.Line().left().dot()
    elm.Capacitor(polar=True).up().reverse().label('C2\n100$\\mu$F\n50V', loc='bot').
↪dot()
    elm.Line().right().hold()
    elm.Resistor().toy(top.end).label('R1\n2.2K\n50V', loc='bot').dot()

    d.move(dx=-d.unit, dy=0)
    elm.Capacitor(polar=True).toy(G.start).flip().label('C1\n 1000$\\mu$F\n50V').dot().
↪idot()
    elm.Line().at(G.start).tox(D.W)
    elm.Line().toy(D.W).dot()

    elm.Resistor().right().at(Q2b.center).label('R2').label('56$\\Omega$ 1W', loc='bot').
↪dot()
    with d.hold():
        elm.Line().toy(top.start).dot()
        elm.Line().tox(Q2.emitter)
    elm.Capacitor(polar=True).toy(G.start).label('C3\n470$\\mu$F\n50V', loc='bot').dot()
    elm.Line().tox(G.start).hold()
    elm.Line().right().dot()
    elm.Resistor().toy(top.center).label('R3\n10K\n1W', loc='bot').dot()
    elm.Line().left().hold()
    elm.Line().right()
    elm.Dot(open=True)
    elm.Gap().toy(G.start).label(['+', '$V_{out}$', '-'])
    elm.Dot(open=True)
    elm.Line().left()
```

4.1.8 5-transistor Operational Transconductance Amplifer (OTA)

Note the use of current labels to show the bias currents.



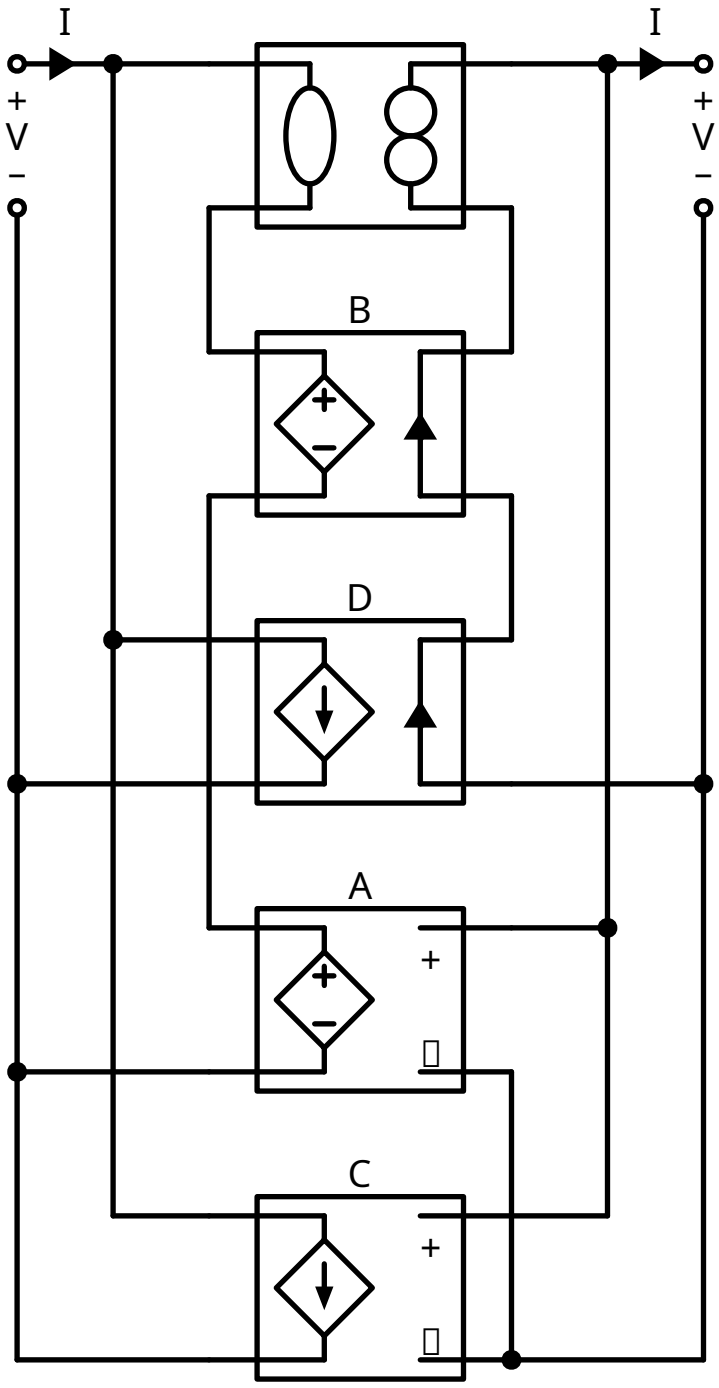

```
with schemdraw.Drawing() as d:
    # tail transistor
    Q1 = elm.AnalogNFet().anchor('source').theta(0).reverse()
    elm.Line().down(0.5)
    ground = d.here
    elm.Ground()

    # input pair
    elm.Line().left(1).at(Q1.drain)
    Q2 = elm.AnalogNFet().anchor('source').theta(0).reverse()

    elm.Dot().at(Q1.drain)
    elm.Line().right(1)
    Q3 = elm.AnalogNFet().anchor('source').theta(0)

    # current mirror
    Q4 = elm.AnalogPFet().anchor('drain').at(Q2.drain).theta(0)
    Q5 = elm.AnalogPFet().anchor('drain').at(Q3.drain).theta(0).reverse()

    elm.Line().right().at(Q4.gate).to(Q5.gate)

    elm.Dot().at(0.5*(Q4.gate + Q5.gate))
    elm.Line().down().toy(Q4.drain)
    elm.Line().left().tox(Q4.drain)
    elm.Dot()

    # vcc connection
    elm.Line().right().at(Q4.source).to(Q5.source)
    elm.Dot().at(0.5*(Q4.source + Q5.source))
    elm.Vdd()

    # bias source
```

(continues on next page)

(continued from previous page)

```
    elm.Line().left(0.25).at(Q1.gate)
    elm.SourceV().down().toy(ground).reverse().scale(0.5).label("Bias")
    elm.Ground()

    # signal labels
    elm.Tag().at(Q2.gate).label("In+").left()
    elm.Tag().at(Q3.gate).label("In−").right()
    elm.Dot().at(Q3.drain)
    elm.Line().right().tox(Q3.gate)
    elm.Tag().right().label("Out").reverse()

    # bias currents
    elm.CurrentLabel(length=1.25, ofst=0.25).at(Q1).label("20μA")
    elm.CurrentLabel(length=1.25, ofst=0.25).at(Q4).label("10μA")
    elm.CurrentLabel(length=1.25, ofst=0.25).at(Q5).label("10μA")
```

4.1.9 Quadruple loop negative feedback amplifier




```
with schemdraw.Drawing() as d:
    # place twoports
    N1 = elm.Nullor().anchor('center')
    T1 = elm.TransimpedanceTransactor(reverse_output=True).reverse().flip().anchor(
↪'center').at([0,-3]).label("B")
    T2 = elm.CurrentTransactor().reverse().flip().anchor('center').at([0,-6]).label("D")
```

(continues on next page)

(continued from previous page)

```
    T3 = elm.VoltageTransactor().reverse().anchor('center').at([0,-9]).label("A")
    T4 = elm.TransadmittanceTransactor(reverse_output=True).reverse().anchor('center').
↪at([0,-12]).label("C")

    ## make connections
    # right side
    elm.Line().at(N1.out_n).to(T1.in_n)
    elm.Line().at(T1.in_p).to(T2.in_n)
    elm.Line().at(T3.in_n).to(T4.in_n)

    elm.Line().right(1).at(N1.out_p)
    pre_out = d.here
    outline = elm.Line().right(1).dot(open=True)
    out = d.here
    elm.Gap().down().label(('+','$V_o$','-')).toy(N1.out_n)
    elm.Line().idot(open=True).down().toy(T4.in_n)
    elm.Line().left().to(T4.in_n)
    elm.Dot()
    elm.CurrentLabelInline(direction='in', ofst=-0.15).at(outline).label('$I_o$')

    elm.Line().at(T2.in_p).right().tox(out)
    elm.Dot()

    elm.Line().right().at(T4.in_p).tox(pre_out)
    elm.Line().up().toy(pre_out)
    elm.Dot()

    elm.Line().right().at(T3.in_p).tox(pre_out)
    elm.Dot()

    # left side
    elm.Line().down().at(N1.in_n).to(T1.out_n)

    elm.Line().up().at(T3.out_p).to(T1.out_p)

    elm.Line().left().at(N1.in_p).length(1)
    pre_in = d.here
    inline = elm.Line().length(1).dot(open=True).left()
    in_node = d.here
    elm.Gap().down().label(('+','$V_i$','-')).toy(N1.in_n)
    elm.Line().idot(open=True).down().toy(T4.out_n)
    elm.Line().right().to(T4.out_n)
    elm.CurrentLabelInline(direction='out', ofst=-0.15).at(inline).label('$I_i$')

    elm.Line().left().at(T2.out_p).tox(in_node)
    elm.Dot()
    elm.Line().left().at(T3.out_n).tox(in_node)
    elm.Dot()

    elm.Line().left().at(T4.out_p).tox(pre_in)
    elm.Line().up().toy(pre_in)
    elm.Dot()
```

(continues on next page)

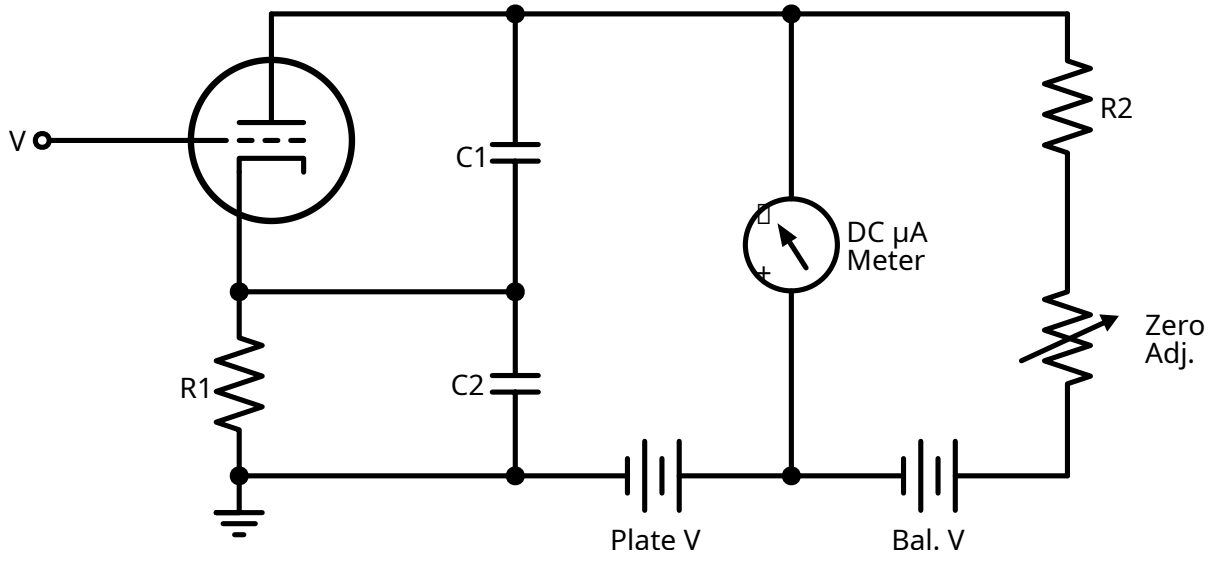
(continued from previous page)

```
    elm.Line().left().at(T2.out_n).tox(pre_in)
    elm.Dot()
```

4.1.10 Vacuum Tube Volt Meters

Schematics from Vacuum Tube Voltmeters, John F. Rider, 1951

```
with schemdraw.Drawing() as d:
    d.config(fontsize=11)
    q = elm.Triode()
    top = elm.Line().up().at(q.anode).length(.5)
    elm.Line().left().at(q.grid).length(2).dot(open=True).label('$V_{in}$', 'left')
    elm.Line().down().at(q.cathode).length(1.3).dot()
    R1 = elm.Resistor().label('R1').length(2).dot()
    elm.Ground()
    elm.Line().right().dot()
    pv = elm.Battery().reverse().label('Plate V', 'bottom').dot()
    elm.Battery().reverse().label('Bal. V', 'bottom')
    elm.ResistorVar().up().flip().label('Zero\nAdj.', 'bottom')
    elm.Resistor().up().label('R2', 'bottom').toy(top.end)
    elm.Line().tox(pv.end).dot()
    elm.MeterArrow().toy(pv.end).reverse().label('DC $\mu$A\nMeter', 'bottom').label('+',
↪'iend', ofst=(.2, -.3)).label('−', 'istart', ofst=(-.15, -.3)).hold()
    elm.Line().tox(pv.start).dot()
    elm.Line().tox(q.anode).hold()
    elm.Capacitor().down().label('C1').toy(R1.start).dot()
    elm.Line().tox(q.cathode).hold()
    elm.Capacitor().toy(pv.start).label('C2')
```

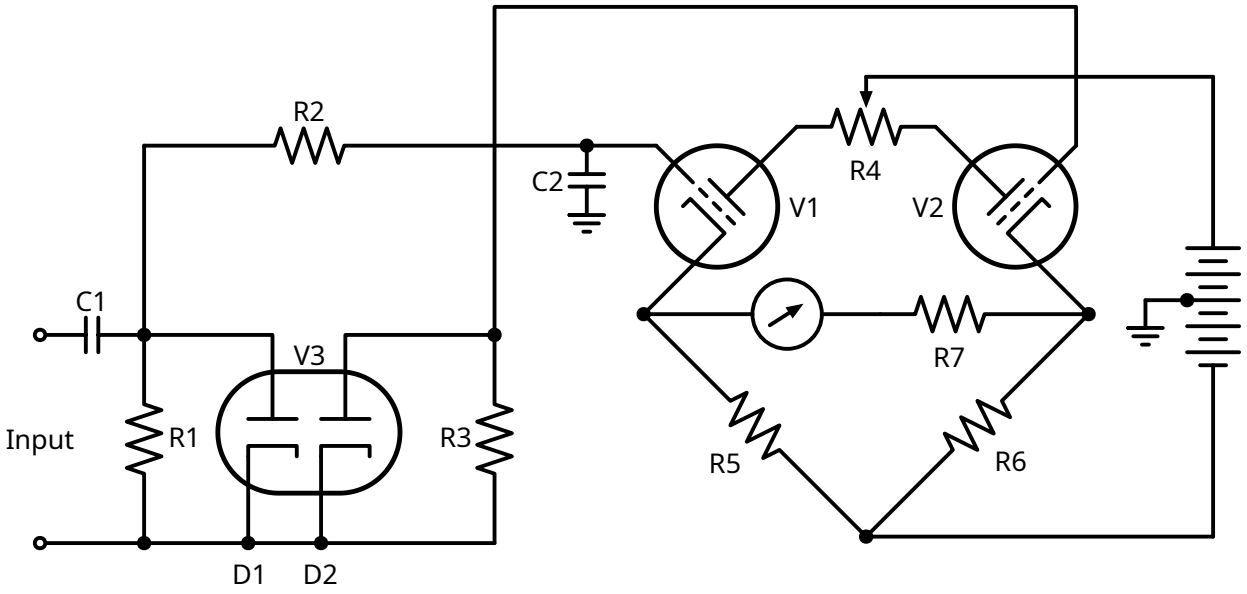



```
with schemdraw.Drawing() as d:
    v3 = elm.DualVacuumTube(grids_left=0, grids_right=0, heater=False).label('V3')
```

(continues on next page)

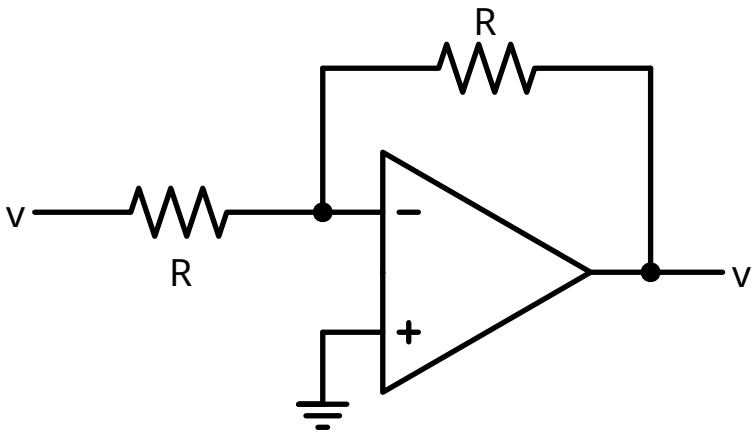
(continued from previous page)

```
    elm.Line().at(v3.cathodeB).down(1.25).dot().label('D2', 'left')
    elm.Line().right(2.5)
    R3 = elm.Resistor().up().label('R3').dot()
    elm.Wire('-|').to(v3.anodeB)
    elm.Line().at(v3.cathodeA).down(1.25).dot().label('D1', 'left')
    elm.Line().tox(v3.cathodeB).hold()
    elm.Line().left(1.5).dot()
    elm.Line().left(1.5).dot(open=True).hold()
    R1 = elm.Resistor().up().label('R1', 'bottom').dot()
    elm.Capacitor().left(1.5).label('C1').dot(open=True)
    elm.Gap().toy(R1.start).label('Input')
    elm.Wire(shape='-|').at(R1.end).to(v3.anodeA).hold()
    v1 = elm.Triode().theta(-45).at((6.5, 3)).label('V1', 'NE')
    elm.Line().theta(135).at(v1.grid).length(.75)
    elm.Line().left(1).dot()
    with d.hold():
        elm.Capacitor().down(1).label('C2')
        elm.Ground(lead=False)
    R2 = elm.Resistor().tox(R1.end).shift(.3).label('R2')
    elm.Line().toy(R1.end)
    d1 = elm.Line().at(v1.cathode_R).theta(-135).length(1.5).dot()
    elm.Line().at(v1.anode).theta(45).length(.75)
    R4 = elm.Potentiometer().right(2).label('R4')
    elm.Line().theta(-45).length(.75)
    v2 = elm.Triode().theta(45).anchor('anode').label('V2', 'NW')
    elm.Line().at(v2.grid_R).theta(45).length(.75)
    elm.Wire('n', k=2).to(R3.end)
    elm.Line().at(v2.cathode).theta(-45).length(1.5).dot()
    R7 = elm.Resistor().shift(.5).left().label('R7', 'bottom')
    elm.MeterArrow().shift(-.3).reverse().tox(d1.end)
    elm.Resistor().theta(-45).label('R5', 'bottom').tox(R4.tap).dot()
    elm.Resistor().to(R7.start).label('R6', 'bottom').hold()
    elm.Line().right(5)
    bat = elm.BatteryDouble().reverse().up().toy(R4.tap)
    elm.Line().tox(R4.tap)
    elm.Line(arrow='o-').at(bat.tap).left(.6)
    elm.Ground()
```



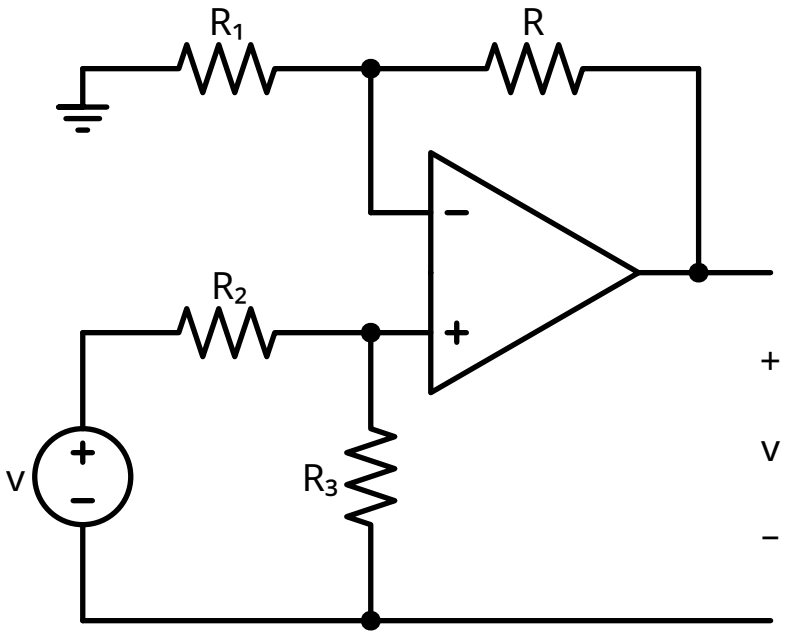
4.2 Opamp Circuits

4.2.1 Inverting Opamp



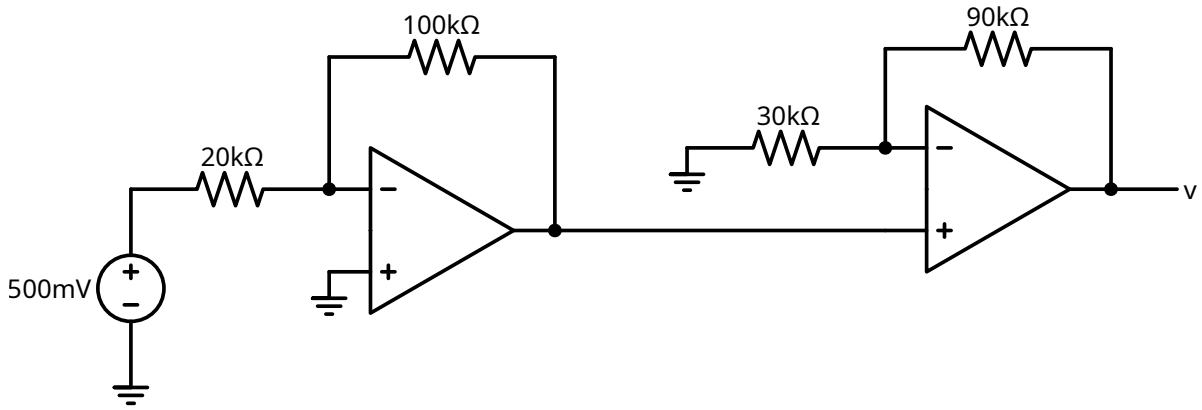

```
with schemdraw.Drawing() as d:
    op = elm.Opamp(leads=True)
    elm.Line().down(d.unit/4).at(op.in2)
    elm.Ground(lead=False)
    Rin = elm.Resistor().at(op.in1).left().idot().label('$R_{in}$', loc='bot').label('$v_
↪{in}$', loc='left')
    elm.Line().up(d.unit/2).at(op.in1)
    elm.Resistor().tox(op.out).label('$R_f$')
    elm.Line().toy(op.out).dot()
    elm.Line().right(d.unit/4).at(op.out).label('$v_{o}$', loc='right')
```

4.2.2 Non-inverting Opamp




```
with schemdraw.Drawing() as d:
    op = elm.Opamp(leads=True)
    out = elm.Line().at(op.out).length(.75)
    elm.Line().up().at(op.in1).length(1.5).dot()
    with d.hold():
        elm.Resistor().left().label('$R_1$')
        elm.Ground()
    elm.Resistor().tox(op.out).label('$R_f$')
    elm.Line().toy(op.out).dot()
    elm.Resistor().left().at(op.in2).idot().label('$R_2$')
    elm.SourceV().down().reverse().label('$v_{in}$')
    elm.Line().right().dot()
    elm.Resistor().up().label('$R_3$').hold()
    elm.Line().tox(out.end)
    elm.Gap().toy(op.out).label(['-','$v_o$','+'])
```

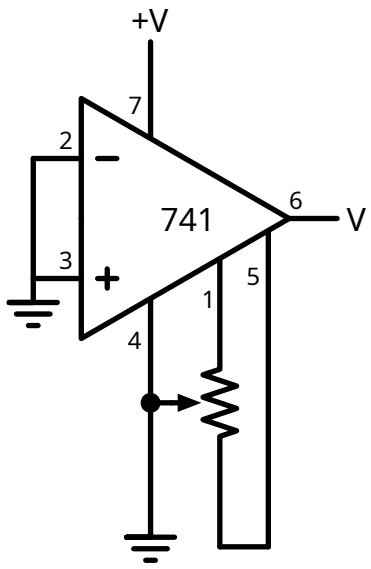
4.2.3 Multi-stage amplifier




```
with schemdraw.Drawing() as d:
    elm.Ground(lead=False)
    elm.SourceV().label('500mV')
    elm.Resistor().right().label(r'20k$\Omega$').dot()
    O1 = elm.Opamp(leads=True).anchor('in1')
    elm.Ground().at(O1.in2)
    elm.Line().up(2).at(O1.in1)
    elm.Resistor().tox(O1.out).label(r'100k$\Omega$')
    elm.Line().toy(O1.out).dot()
    elm.Line().right(5).at(O1.out)
    O2 = elm.Opamp(leads=True).anchor('in2')
    elm.Resistor().left().at(O2.in1).idot().label(r'30k$\Omega$')
    elm.Ground()
    elm.Line().up(1.5).at(O2.in1)
    elm.Resistor().tox(O2.out).label(r'90k$\Omega$')
    elm.Line().toy(O2.out).dot()
    elm.Line().right(1).at(O2.out).label('$v_{out}$', loc='rgt')
```

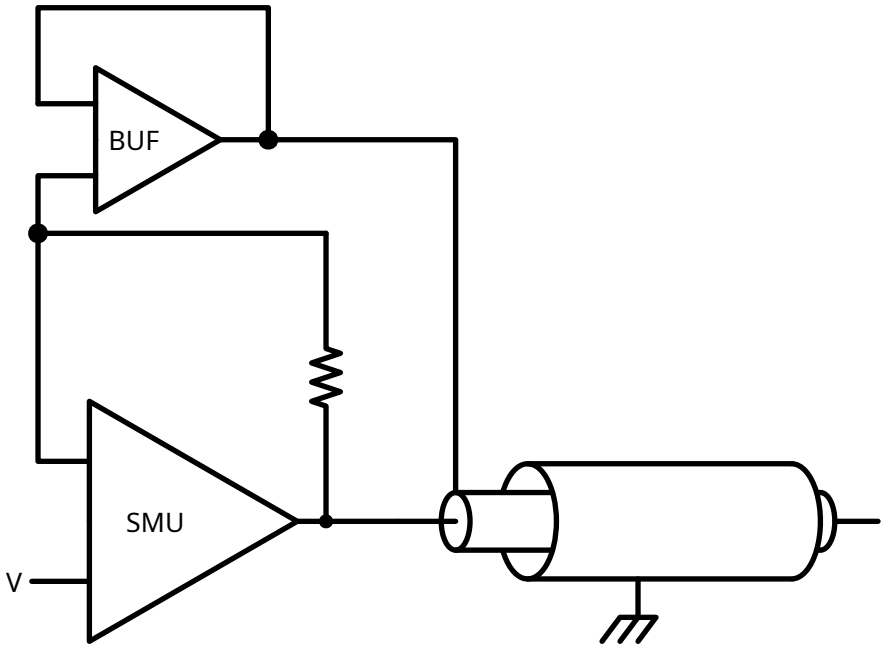
4.2.4 Opamp pin labeling

This example shows how to label pin numbers on a 741 opamp, and connect to the offset anchors.




```
with schemdraw.Drawing() as d:
    d.config(fontsize=12)
    op = (elm.Opamp()
        .label('741', loc='center', ofst=0)
        .label('1', 'n1')
        .label('5', 'n1a')
        .label('4', 'vs')
        .label('7', 'vd')
        .label('2', 'in1')
        .label('3', 'in2')
        .label('6', 'out')
    )
    elm.Line().left(.5).at(op.in1)
    elm.Line().down(d.unit/2)
    elm.Ground(lead=False)
    elm.Line().left(.5).at(op.in2)
    elm.Line().right(.5).at(op.out).label('$V_o$', 'right')
    elm.Line().up(1).at(op.vd).label('$+V_s$', 'right')
    trim = elm.Potentiometer().down().at(op.n1).flip().scale(0.7)
    elm.Line().tox(op.n1a)
    elm.Line().up().to(op.n1a)
    elm.Line().at(trim.tap).tox(op.vs).dot()
    with d.hold():
        elm.Line().down(d.unit/3)
        elm.Ground()
    elm.Line().toy(op.vs)
```

4.2.5 Triaxial Cable Driver




```
with schemdraw.Drawing() as d:
    d.config(fontsize=10)
    elm.Line().length(d.unit/5).label('V', 'left')
    smu = (elm.Opamp(sign=False).anchor('in2')
              .label('SMU', 'center', ofst=[-.4, 0], halign='center', valign='center'))
    elm.Line().at(smu.out).length(.3)
    with d.hold():
        elm.Line().length(d.unit/4)
        triax = elm.Triax(length=5, shieldofststart=.75)
    elm.Resistor().up().scale(0.6).idot()
    elm.Line().left().dot()
    elm.Wire('|-').to(smu.in1).hold()
    elm.Wire('|-').delta(d.unit/5, d.unit/5)
    buf = (elm.Opamp(sign=False).anchor('in2').scale(0.6)
              .label('BUF', 'center', ofst=(-.4, 0), halign='center', valign='center'))
    elm.Line().left(d.unit/5).at(buf.in1)
    elm.Wire('n').to(buf.out, dx=.5).dot()
    elm.Wire('-|').at(buf.out).to(triax.guardstart_top)
    elm.GroundChassis().at(triax.shieldcenter)
```

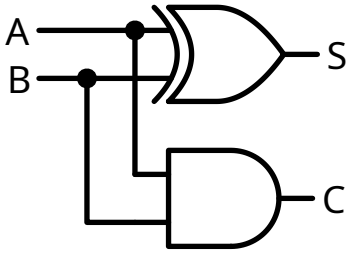
4.3 Digital Logic

Logic gate definitions are in the *schemdraw.logic.logic* module. Here it was imported with

```
from schemdraw import logic
```

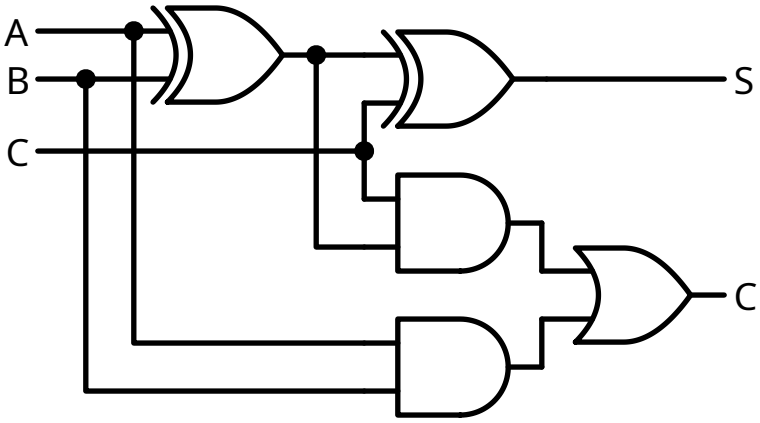
4.3.1 Half Adder

Notice the half and full adders set the drawing unit to 0.5 so the lines aren't quite as long and look better with logic gates.




```
with schemdraw.Drawing() as d:
    d.config(unit=0.5)
    S = logic.Xor().label('S', 'right')
    logic.Line().left(d.unit*2).at(S.in1).idot().label('A', 'left')
    B = logic.Line().left().at(S.in2).dot()
    logic.Line().left().label('B', 'left')
    logic.Line().down(d.unit*3).at(S.in1)
    C = logic.And().right().anchor('in1').label('C', 'right')
    logic.Wire('|-').at(B.end).to(C.in2)
```

4.3.2 Full Adder




```
with schemdraw.Drawing() as d:
    d.config(unit=0.5)
    X1 = logic.Xor()
```

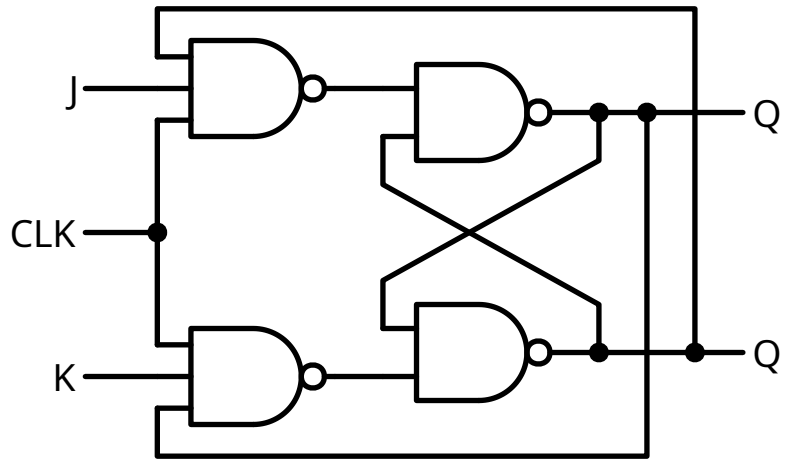
(continues on next page)

(continued from previous page)

```
    A = logic.Line().left(d.unit*2).at(X1.in1).idot().label('A', 'left')
    B = logic.Line().left().at(X1.in2).dot()
    logic.Line().left().label('B', 'left')
    logic.Line().right().at(X1.out).idot()
    X2 = logic.Xor().anchor('in1')
    C = logic.Line().down(d.unit*2).at(X2.in2)
    with d.hold():
        logic.Dot().at(C.center)
        logic.Line().tox(A.end).label('C$_{in}$', 'left')
    A1 = logic.And().right().anchor('in1')
    logic.Wire('-|').at(A1.in2).to(X1.out)
    d.move_from(A1.in2, dy=-d.unit*2)
    A2 = logic.And().right().anchor('in1')
    logic.Wire('-|').at(A2.in1).to(A.start)
    logic.Wire('-|').at(A2.in2).to(B.end)
    d.move_from(A1.out, dy=-(A1.out.y-A2.out.y)/2)
    O1 = logic.Or().right().label('C$_{out}$', 'right')
    logic.Line().at(A1.out).toy(O1.in1)
    logic.Line().at(A2.out).toy(O1.in2)
    logic.Line().at(X2.out).tox(O1.out).label('S', 'right')
```

4.3.3 J-K Flip Flop

Note the use of the LaTeX command **overline{Q}** in the label to draw a bar over the inverting output label.



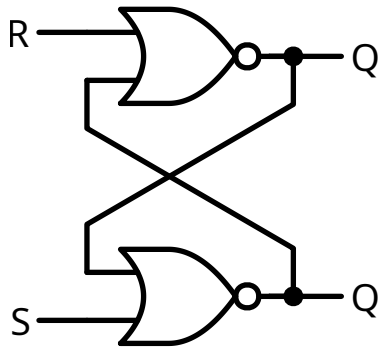

```
with schemdraw.Drawing() as d:
    # Two front gates (SR latch)
    G1 = logic.Nand(leadout=.75).anchor('in1')
    logic.Line().length(d.unit/2).label('Q', 'right')
    d.move_from(G1.in1, dy=-2.5)
    G2 = logic.Nand(leadout=.75).anchor('in1')
    logic.Line().length(d.unit/2).label(r'$\overline{Q}$', 'right')
    logic.Wire('N', k=.5).at(G2.in1).to(G1.out).dot()
    logic.Wire('N', k=.5).at(G1.in2).to(G2.out).dot()
```

(continues on next page)

(continued from previous page)

```
    # Two back gates
    logic.Line().left(d.unit/6).at(G1.in1)
    J = logic.Nand(inputs=3).anchor('out').right()
    logic.Wire('n', k=.5).at(J.in1).to(G2.out, dx=1).dot()
    logic.Line().left(d.unit/4).at(J.in2).label('J', 'left')
    logic.Line().left(d.unit/6).at(G2.in2)
    K = logic.Nand(inputs=3).right().anchor('out')
    logic.Wire('n', k=-.5).at(K.in3).to(G1.out, dx=.5).dot()
    logic.Line().left(d.unit/4).at(K.in2).label('K', 'left')
    C = logic.Line().at(J.in3).toy(K.in1)
    logic.Dot().at(C.center)
    logic.Line().left(d.unit/4).label('CLK', 'left')
```

4.3.4 S-R Latch (Gates)




```
with schemdraw.Drawing() as d:
    g1 = logic.Nor()
    d.move_from(g1.in1, dy=-2.5)
    g2 = logic.Nor().anchor('in1')
    g1out = logic.Line().right(.25).at(g1.out)
    logic.Wire('N', k=.5).at(g2.in1).to(g1out.end).dot()
    g2out = logic.Line().right(.25).at(g2.out)
    logic.Wire('N', k=.5).at(g1.in2).to(g2out.end).dot()
    logic.Line().at(g1.in1).left(.5).label('R', 'left')
    logic.Line().at(g2.in2).left(.5).label('S', 'left')
    logic.Line().at(g1.out).right(.75).label('Q', 'right')
    logic.Line().at(g2.out).right(.75).label(r'$\overline{Q}$', 'right')
```

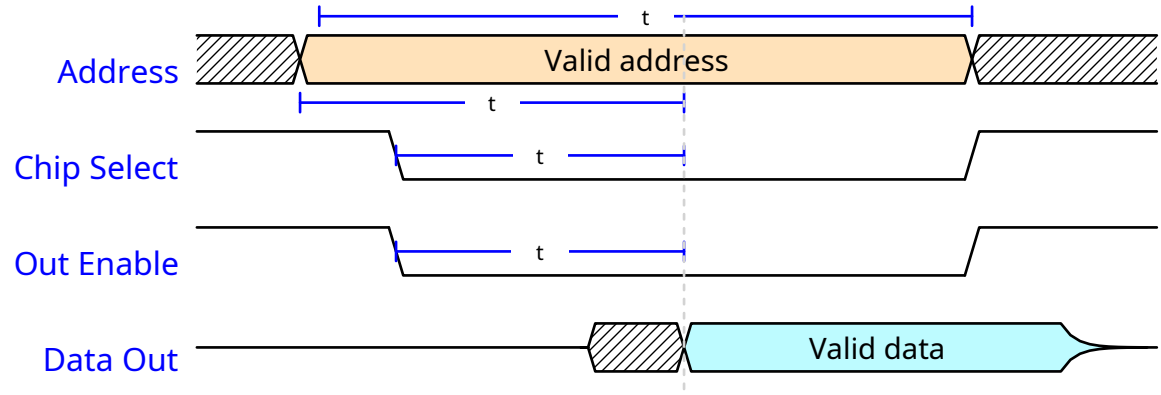
4.4 Timing Diagrams

Timing diagrams, based on WaveDrom, are drawn using the *schemdraw.logic.timing.TimingDiagram* class.

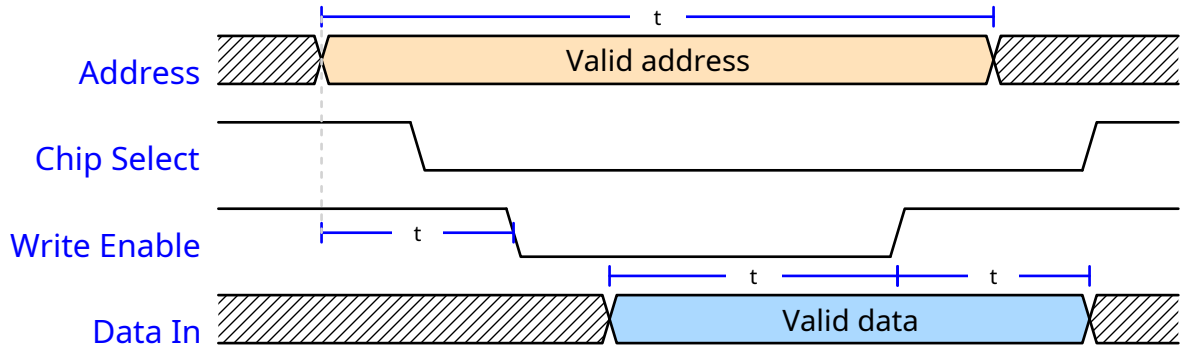
```
from schemdraw import logic
```

4.4.1 SRAM read/write cycle

The SRAM examples make use of Schemdraw's extended 'edge' notation for labeling timings just above and below the wave.



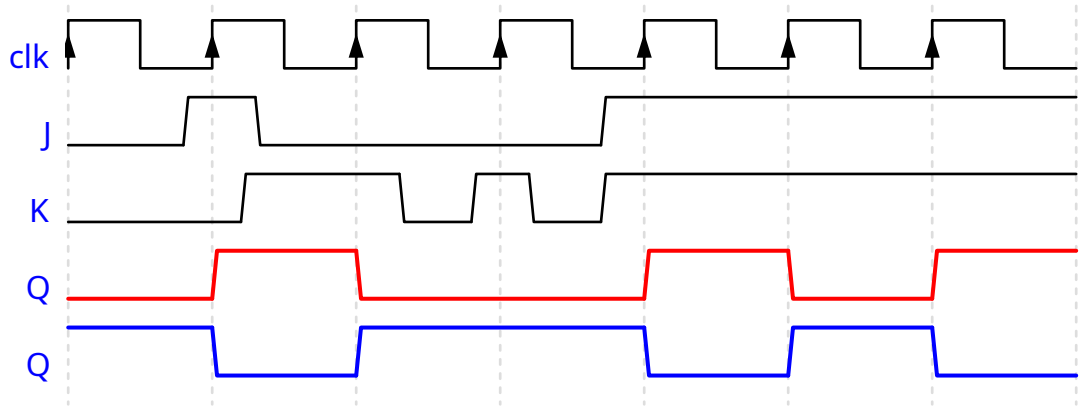

```
logic.TimingDiagram(
    {'signal': [
        {'name': 'Address',     'wave': 'x4......x.', 'data': ['Valid address']},
        {'name': 'Chip Select', 'wave': '1.0.....1.'},
        {'name': 'Out Enable',  'wave': '1.0.....1.'},
        {'name': 'Data Out',    'wave': 'z...x6...z', 'data': ['Valid data']},
    ],
     'edge': ['[0^:1.2]+[0^:8] $t_{WC}$',
              '[0v:1]+[0v:5] $t_{AQ}$',
              '[1:2]+[1:5] $t_{EQ}$',
              '[2:2]+[2:5] $t_{GQ}$',
              '[0^:5]-[3v:5]{lightgray,:}',
             ]
    }, ygap=.5, grid=False)
```



```
logic.TimingDiagram(
    {'signal': [
        {'name': 'Address',      'wave': 'x4......x.', 'data': ['Valid address']},
        {'name': 'Chip Select',  'wave': '1.0......1'},
        {'name': 'Write Enable', 'wave': '1..0...1..'},
        {'name': 'Data In',      'wave': 'x...5....x', 'data': ['Valid data']},
    ],
     'edge': ['[0^:1]+[0^:8] $t_{WC}$',
              '[2:1]+[2:3] $t_{SA}$',
              '[3^:4]+[3^:7] $t_{WD}$',
              '[3^:7]+[3^:9] $t_{HD}$',
              '[0^:1]-[2:1]{lightgray,:}'],
    }, ygap=.4, grid=False)
```

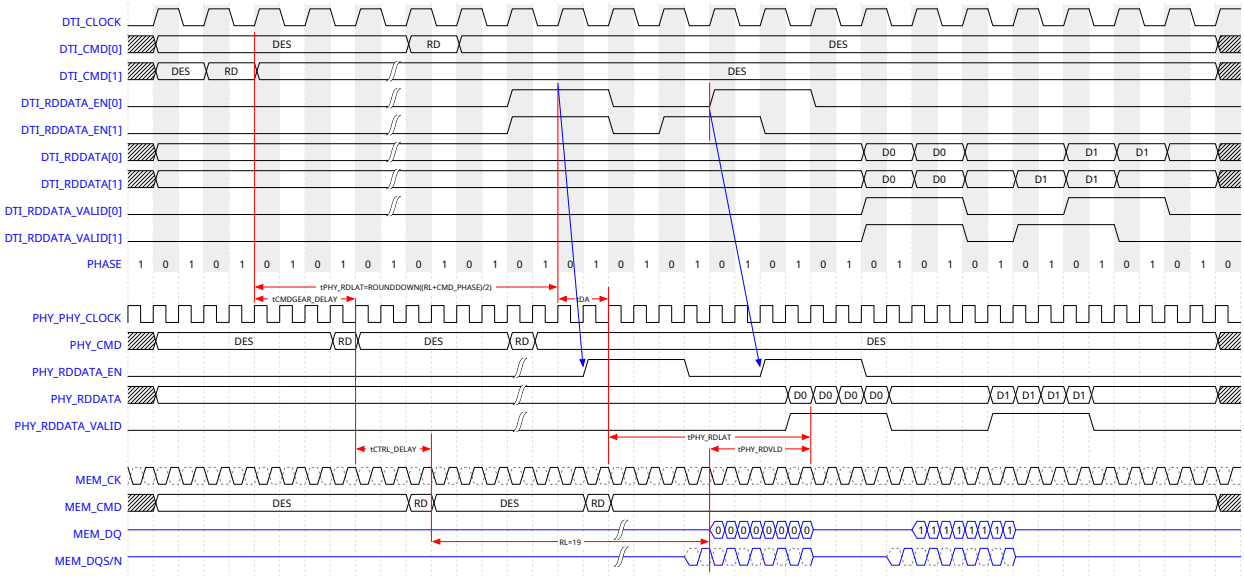
4.4.2 J-K Flip Flop

Timing diagram for a J-K flip flop taken from here. Notice the use of the *async* dictionary parameter on the J and K signals, and the color parameters for the output signals.




```
logic.TimingDiagram(
    {'signal': [
        {'name': 'clk', 'wave': 'P......'},
        {'name': 'J', 'wave': '0101', 'async': [0, .8, 1.3, 3.7, 7]},
        {'name': 'K', 'wave': '010101', 'async': [0, 1.2, 2.3, 2.8, 3.2, 3.7, 7]},
        {'name': 'Q', 'wave': '010.101', 'color': 'red', 'lw': 1.5},
        {'name': r'$\overline{Q}$', 'wave': '101.010', 'color': 'blue', 'lw': 1.5}],
    'config': {'hscale': 1.5}}, risetime=.05)
```

4.4.3 SDRAM Timing Diagram




```
logic.TimingDiagram.from_json('''
{"signal": [
    {"name": "DTI_CLOCK", "wave": "0101010101010101010101010101010101010101010101010101"},
    {"name": "DTI_CMD[0]", "wave": "x2.........2.2.............................x", "data
↪": ["DES", "RD", "DES"]},
    {"name": "DTI_CMD[1]",       "wave": "x2.2.2....|.............................x",
↪"data": ["DES", "RD", "DES"]},
    {"name": "DTI_RDDATA_EN[0]", "wave": "0.........|....1...0...1...0.............",
↪"data": ["DES", "RD", "DES"]},
    {"name": "DTI_RDDATA_EN[1]", "wave": "0.........|....1...0.1...0.................",
↪"data": ["DES", "RD", "DES"]},
    {"name": "DTI_RDDATA[0]",       "wave": "x2........|.................2.2.2...2.2.2.x
↪", "data": ["", "D0", "D0", "", "D1", "D1"]},
    {"name": "DTI_RDDATA[1]",       "wave": "x2........|.................2.2.2.2.2.2...x
↪", "data": ["", "D0", "D0", "", "D1", "D1"]},
    {"name": "DTI_RDDATA_VALID[0]", "wave": "0.........|.................1...0...1...0..
↪"},
    {"name": "DTI_RDDATA_VALID[1]", "wave": "0.........................1...0.1...0....
↪"},
    {"name": "PHASE", "data": "{1 0}" },
    {},
    {"name": "PHY_PHY_CLOCK", "wave": "p.........................................."},
    {"name": "PHY_CMD",       "wave": "x2......22.....22.........................x",
↪"data": ["DES", "RD", "DES", "RD", "DES"]},
    {"name": "PHY_RDDATA_EN", "wave": "0.............|..1...0..1...0.............."},
    {"name": "PHY_RDDATA",       "wave": "x2.............|.........22222...22222....x",
↪"data": ["", "D0", "D0", "D0", "D0", "", "D1", "D1", "D1", "D1", ""]},
    {"name": "PHY_RDDATA_VALID", "wave": "0.............|.........1...0...1...0.....",
↪"data": ["", "D0", "D0", "D0", "D0", "", "D1", "D1", "D1", "D1", ""]},
    {},
    {"name": "MEM_CK",    "wave": "Q.........................................."},
    {"name": "MEM_CMD",   "wave": "x2.........22.....22.......................x", "data
```

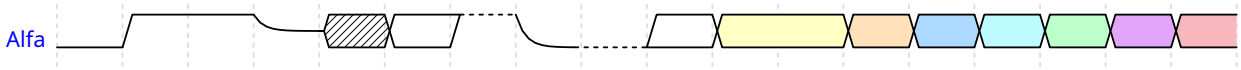
(continues on next page)

(continued from previous page)

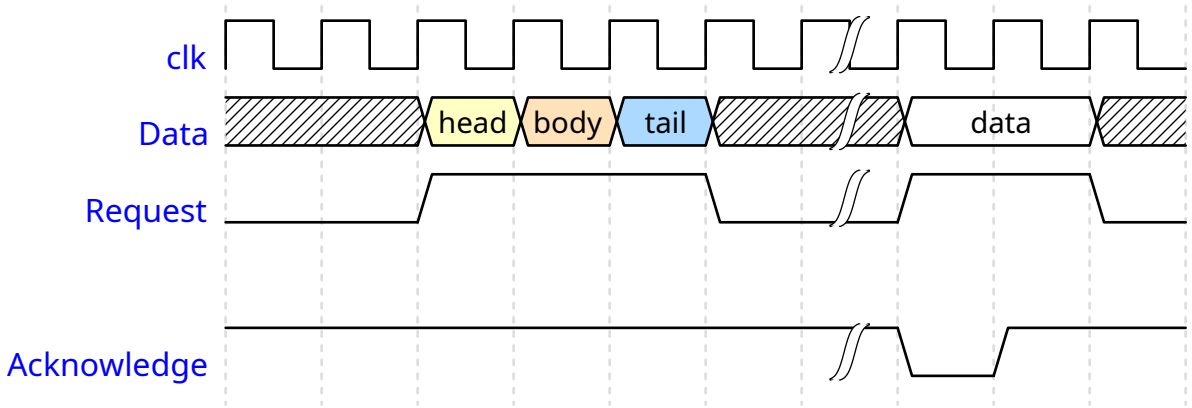
```
↪": ["DES", "RD", "DES", "RD"]},
    {"name": "MEM_DQ",    "wave": "z.................|...bbbbz...bbbbz........", "data
↪": ["0", "0", "0", "0", "0", "0", "0", "0", "1", "1", "1", "1", "1", "1", "1", "1"],
↪"color": "blue"},
    {"name": "MEM_DQS/N", "wave": "z.................|..Q....z..Q....z........", "color
↪": "blue"},

],
"shade": [
    "odd 0:9 #eee",
],
"edge": [
    "[0v:5]-[10v:5]{red}",
    "[2v:17]-[10v:17]{red}",
    "[10:5]<->[10:17]{red} tPHY_RDLAT=ROUNDDOWN((RL+CMD_PHASE)/2)",
    "[11^:5]<->[11^:9]{red} tCMDGEAR_DELAY",
    "[11^:9]-[16v:9]{red}",
    "[11^:17]<->[11^:19]{red} tDA",
    "[10:19]-[16v:19]{red}",
    "[15v:19]<->[15v:27]{red} tPHY_RDLAT",
    "[15^:27]-[16v:27]{red}",
    "[16:23]<->[16:27]{red} tPHY_RDVLD",
    "[16:23]-[20v:23]{red}",
    "[20^:23]<->[20^:12]{red} RL=19",
    "[20^:12]-[16^:12]{red}",
    "[16:12]<->[16:9]{red} tCTRL_DELAY",

    "[3^:17]->[13:18]{blue}",
    "[4^:23]->[13:25]{blue}",
    "[3^:23]-[4v:23]{red}",

    ],
"config": {"hscale": .75}
}''',

nodealign='clock'
)
```

4.4.4 Tutorial Examples

These examples were copied from WaveDrom Tutorial. They use the *from_json* class method so the examples can be pasted directly as a string. Otherwise, the setup must be converted to a proper Python dictionary.



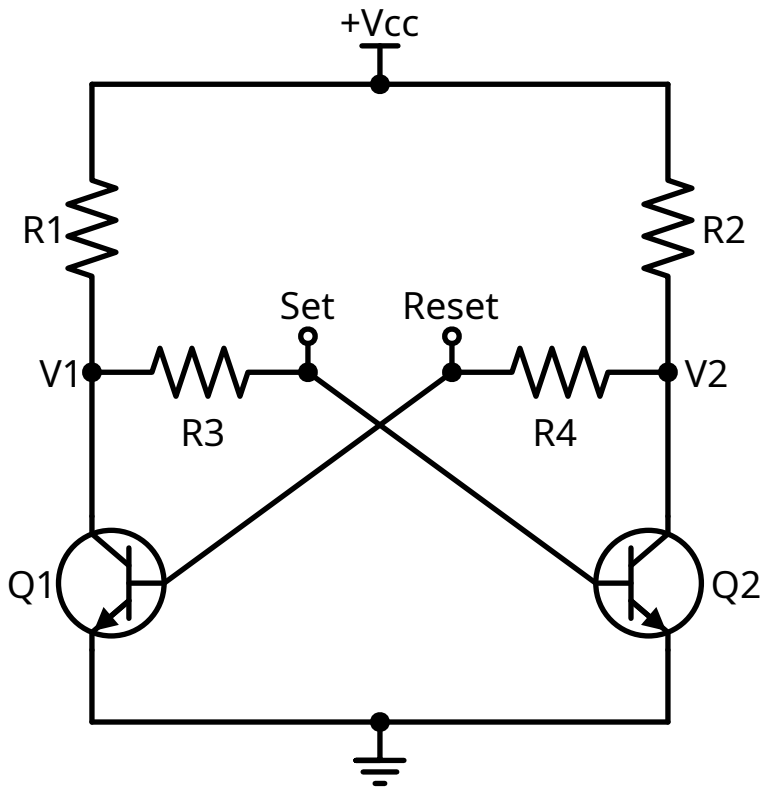

```
logic.TimingDiagram.from_json('''{ signal: [{ name: "Alfa", wave: "01.zx=ud.23.456789" }
↪] }''')
```




```
logic.TimingDiagram.from_json('''{ signal: [
  { name: "clk",         wave: "p.....|..." },
  { name: "Data",        wave: "x.345x|=.x", data: ["head", "body", "tail", "data"] },
  { name: "Request",     wave: "0.1..0|1.0" },
  {},
  { name: "Acknowledge", wave: "1.....|01." }
  ]}''')
```

4.5 Solid State

4.5.1 S-R Latch (Transistors)



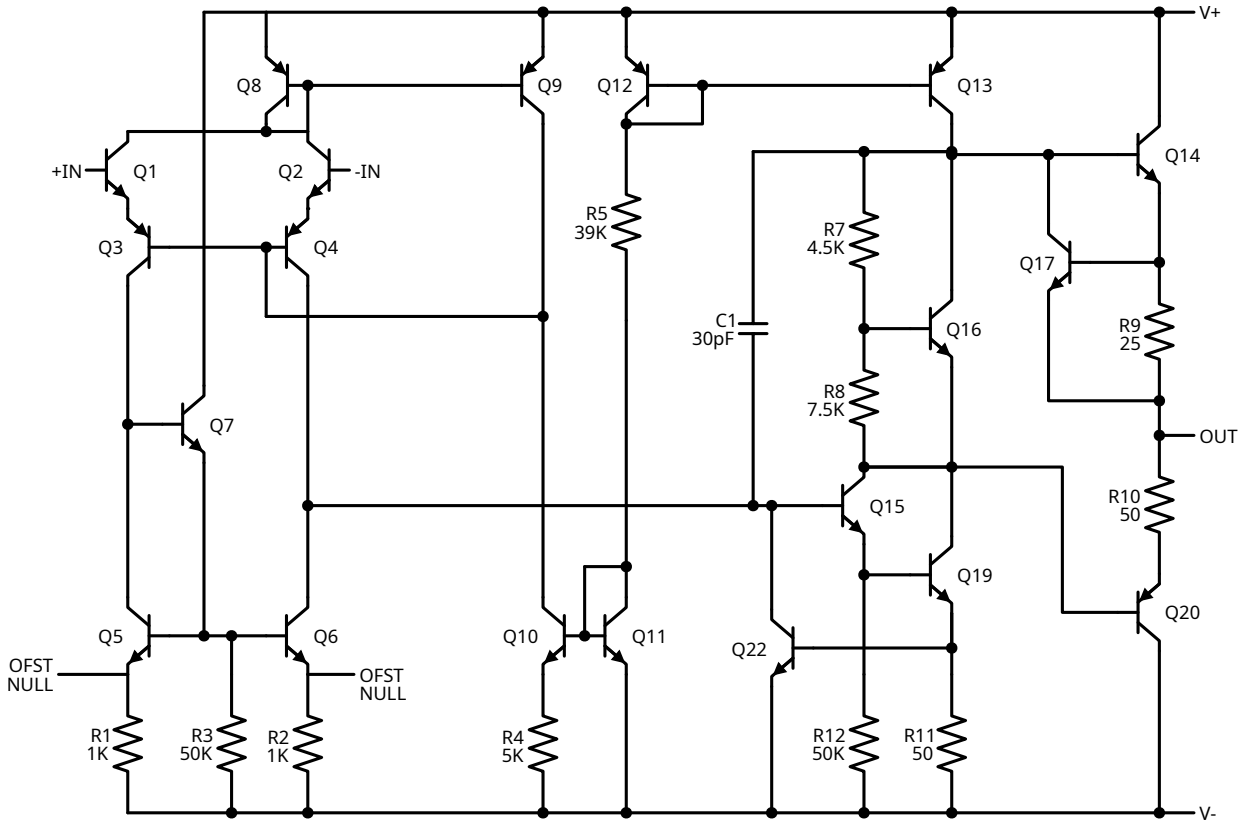

```
with schemdraw.Drawing() as d:
    Q1 = elm.BjtNpn(circle=True).reverse().label('Q1', 'left')
    Q2 = elm.BjtNpn(circle=True).at((d.unit*2, 0)).label('Q2')
    elm.Line().up(d.unit/2).at(Q1.collector)

    R1 = elm.Resistor().up().label('R1').hold()
    elm.Dot().label('V1', 'left')
    elm.Resistor().right(d.unit*.75).label('R3', 'bottom').dot()
    elm.Line().up(d.unit/8).dot(open=True).label('Set', 'right').hold()
    elm.Line().to(Q2.base)

    elm.Line().up(d.unit/2).at(Q2.collector)
    elm.Dot().label('V2', 'right')
    R2 = elm.Resistor().up().label('R2', 'bottom').hold()
    elm.Resistor().left(d.unit*.75).label('R4', 'bottom').dot()
    elm.Line().up(d.unit/8).dot(open=True).label('Reset', 'right').hold()
    elm.Line().to(Q1.base)

    elm.Line().down(d.unit/4).at(Q1.emitter)
    BOT = elm.Line().tox(Q2.emitter)
    elm.Line().to(Q2.emitter)
    elm.Dot().at(BOT.center)
```

(continues on next page)

(continued from previous page)

```
    elm.Ground().at(BOT.center)

    TOP = elm.Line().endpoints(R1.end, R2.end)
    elm.Dot().at(TOP.center)
    elm.Vdd().at(TOP.center).label('+Vcc')
```

4.5.2 741 Opamp Internal Schematic




```
with schemdraw.Drawing() as d:
    d.config(fontsize=12, unit=2.5)
    Q1 = elm.BjtNpn().label('Q1').label('+IN', 'left')
    Q3 = elm.BjtPnp().left().at(Q1.emitter).anchor('emitter').flip().label('Q3', 'left')
    elm.Line().down().at(Q3.collector).dot()
    with d.hold():
        elm.Line().right(d.unit/4)
        Q7 = elm.BjtNpn().anchor('base').label('Q7')
    elm.Line().down(d.unit*1.25)
    Q5 = elm.BjtNpn().left().flip().anchor('collector').label('Q5', 'left')
    elm.Line().left(d.unit/2).at(Q5.emitter).label('OFST\nNULL', 'left').flip()
    elm.Resistor().down().at(Q5.emitter).label('R1\n1K')
    elm.Line().right(d.unit*.75).dot()
    R3 = elm.Resistor().up().label('R3\n50K')
    elm.Line().toy(Q5.base).dot()
```

(continues on next page)

(continued from previous page)

```
    with d.hold():
        elm.Line().left().to(Q5.base)
        elm.Line().at(Q7.emitter).toy(Q5.base).dot()
    elm.Line().right(d.unit/4)
    Q6 = elm.BjtNpn().anchor('base').label('Q6')
    elm.Line().at(Q6.emitter).length(d.unit/3).label('\nOFST\nNULL', 'right').hold()
    elm.Resistor().down().at(Q6.emitter).label('R2\n1K').dot()

    elm.Line().at(Q6.collector).toy(Q3.collector)
    Q4 = elm.BjtPnp().right().anchor('collector').label('Q4')
    elm.Line().at(Q4.base).tox(Q3.base)
    elm.Line().at(Q4.emitter).toy(Q1.emitter)
    Q2 = elm.BjtNpn().left().flip().anchor('emitter').label('Q2', 'left').label('$-$IN',
↪'right')
    elm.Line().up(d.unit/3).at(Q2.collector).dot()
    Q8 = elm.BjtPnp().left().flip().anchor('base').label('Q8', 'left')
    elm.Line().at(Q8.collector).toy(Q2.collector).dot()
    elm.Line().at(Q2.collector).tox(Q1.collector)
    elm.Line().up(d.unit/4).at(Q8.emitter)
    top = elm.Line().tox(Q7.collector)
    elm.Line().toy(Q7.collector)

    elm.Line().right(d.unit*2).at(top.start)
    elm.Line().down(d.unit/4)
    Q9 = elm.BjtPnp().right().anchor('emitter').label('Q9', ofst=-.1)
    elm.Line().at(Q9.base).tox(Q8.base)
    elm.Dot().at(Q4.base)
    elm.Line().down(d.unit/2).at(Q4.base)
    elm.Line().tox(Q9.collector).dot()
    elm.Line().at(Q9.collector).toy(Q6.collector)
    Q10 = elm.BjtNpn().left().flip().anchor('collector').label('Q10', 'left')
    elm.Resistor().at(Q10.emitter).toy(R3.start).label('R4\n5K').dot()

    Q11 = elm.BjtNpn().right().at(Q10.base).anchor('base').label('Q11')
    elm.Dot().at(Q11.base)
    elm.Line().up(d.unit/2)
    elm.Line().tox(Q11.collector).dot()
    elm.Line().at(Q11.emitter).toy(R3.start).dot()
    elm.Line().up(d.unit*2).at(Q11.collector)
    elm.Resistor().toy(Q9.collector).label('R5\n39K')
    Q12 = elm.BjtPnp().left().flip().anchor('collector').label('Q12', 'left', ofst=-.1)
    elm.Line().up(d.unit/4).at(Q12.emitter).dot()
    elm.Line().tox(Q9.emitter).dot()
    elm.Line().right(d.unit/4).at(Q12.base).dot()
    elm.Wire('|-').to(Q12.collector).dot().hold()
    elm.Line().right(d.unit*1.5)
    Q13 = elm.BjtPnp().anchor('base').label('Q13')
    elm.Line().up(d.unit/4).dot()
    elm.Line().tox(Q12.emitter)
    K = elm.Line().down(d.unit/5).at(Q13.collector).dot()
    elm.Line().down()
    Q16 = elm.BjtNpn().right().anchor('collector').label('Q16', ofst=-.1)
```

(continues on next page)

(continued from previous page)

```
    elm.Line().left(d.unit/3).at(Q16.base).dot()
    R7 = elm.Resistor().up().toy(K.end).label('R7\n4.5K').dot()
    elm.Line().tox(Q13.collector).hold()
    R8 = elm.Resistor().down().at(R7.start).label('R8\n7.5K').dot()
    elm.Line().tox(Q16.emitter)
    J = elm.Dot()
    elm.Line().toy(Q16.emitter)
    Q15 = elm.BjtNpn().right().at(R8.end).anchor('collector').label('Q15')
    elm.Line().left(d.unit/2).at(Q15.base).dot()
    C1 = elm.Capacitor().toy(R7.end).label('C1\n30pF')
    elm.Line().tox(Q13.collector)
    elm.Line().at(C1.start).tox(Q6.collector).dot()
    elm.Line().down(d.unit/2).at(J.center)
    Q19 = elm.BjtNpn().right().anchor('collector').label('Q19')
    elm.Line().at(Q19.base).tox(Q15.emitter).dot()
    elm.Line().toy(Q15.emitter).hold()
    elm.Line().down(d.unit/4).at(Q19.emitter).dot()
    elm.Line().left()
    Q22 = elm.BjtNpn().left().anchor('base').flip().label('Q22', 'left')
    elm.Line().at(Q22.collector).toy(Q15.base).dot()
    elm.Line().at(Q22.emitter).toy(R3.start).dot()
    elm.Line().tox(R3.start).hold()
    elm.Line().tox(Q15.emitter).dot()
    with d.hold():
        elm.Resistor().up().label('R12\n50K')
        elm.Line().toy(Q19.base)
    elm.Line().tox(Q19.emitter).dot()
    R11 = elm.Resistor().up().label('R11\n50')
    elm.Line().toy(Q19.emitter)

    elm.Line().up(d.unit/4).at(Q13.emitter)
    elm.Line().right(d.unit*1.5).dot()
    elm.Line().length(d.unit/4).label('V+', 'right').hold()
    elm.Line().down(d.unit*.75)
    Q14 = elm.BjtNpn().right().anchor('collector').label('Q14')
    elm.Line().left(d.unit/2).at(Q14.base)
    with d.hold():
        elm.Line().down(d.unit/2).idot()
        Q17 = elm.BjtNpn().left().anchor('collector').flip().label('Q17', 'left', ofst=-.
↪1)
        elm.Line().at(Q17.base).tox(Q14.emitter).dot()
        J = elm.Line().toy(Q14.emitter)
    elm.Line().tox(Q13.collector).dot()
    elm.Resistor().down().at(J.start).label('R9\n25').dot()
    elm.Wire('-|').to(Q17.emitter).hold()
    elm.Line().down(d.unit/4).dot()
    elm.Line().right(d.unit/4).label('OUT', 'right').hold()
    elm.Resistor().down().label('R10\n50')
    Q20 = elm.BjtPnp().right().anchor('emitter').label('Q20')
    elm.Wire('c', k=-1).at(Q20.base).to(Q15.collector)
    elm.Line().at(Q20.collector).toy(R3.start).dot()
    elm.Line().right(d.unit/4).label('V-', 'right').hold()
```

(continues on next page)

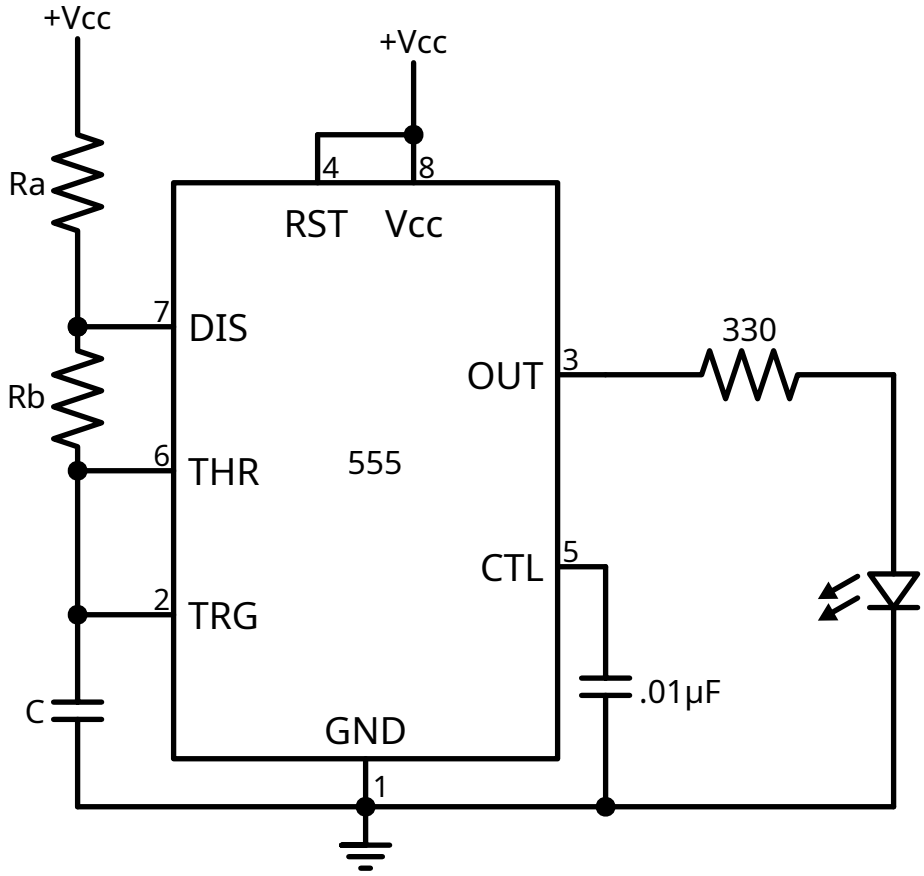
(continued from previous page)

```
    elm.Line().tox(R11.start)
```

4.6 Integrated Circuits

4.6.1 555 LED Blinker Circuit

Using the *schemdraw.elements.intcircuits.Ic* class to define a custom integrated circuit.



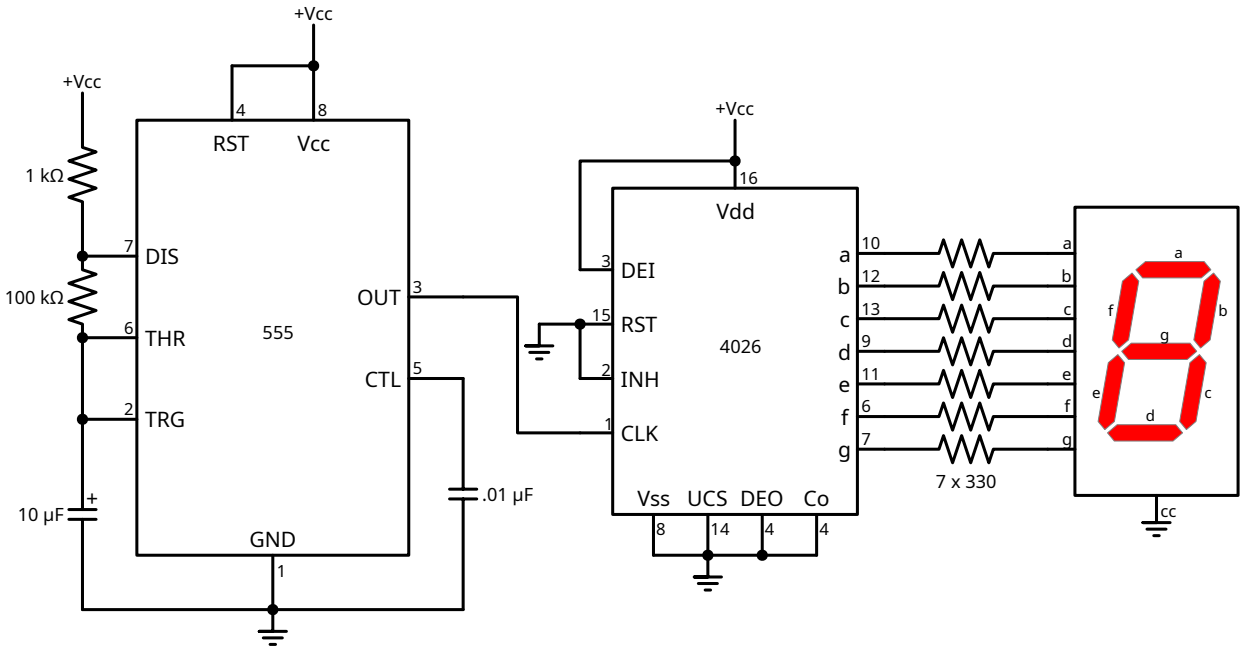

```
with schemdraw.Drawing() as d:
    d.config(fontsize=12)
    T = (elm.Ic()
         .side('L', spacing=1.5, pad=1.5, leadlen=1)
         .side('R', spacing=2)
         .side('T', pad=1.5, spacing=1)
         .pin(name='TRG', side='left', pin='2')
         .pin(name='THR', side='left', pin='6')
         .pin(name='DIS', side='left', pin='7')
         .pin(name='CTL', side='right', pin='5')
         .pin(name='OUT', side='right', pin='3')
         .pin(name='RST', side='top', pin='4')
         .pin(name='Vcc', side='top', pin='8')
```

(continues on next page)

(continued from previous page)

```
        .pin(name='GND', side='bot', pin='1')
        .label('555'))
    BOT = elm.Ground().at(T.GND)
    elm.Dot()
    elm.Resistor().endpoints(T.DIS, T.THR).label('Rb').idot()
    elm.Resistor().up().at(T.DIS).label('Ra').label('+Vcc', 'right')
    elm.Line().endpoints(T.THR, T.TRG)
    elm.Capacitor().at(T.TRG).toy(BOT.start).label('C')
    elm.Line().tox(BOT.start)
    elm.Capacitor().at(T.CTL).toy(BOT.start).label(r'.01$\mu$F', 'bottom').dot()
    elm.Dot().at(T.DIS)
    elm.Dot().at(T.THR)
    elm.Dot().at(T.TRG)
    elm.Line().endpoints(T.RST,T.Vcc).dot()
    elm.Line().up(d.unit/4).label('+Vcc', 'right')
    elm.Resistor().right().at(T.OUT).label('330')
    elm.LED().flip().toy(BOT.start)
    elm.Line().tox(BOT.start)
```

4.6.2 Seven-Segment Display Counter




```
with schemdraw.Drawing() as d:
    d.config(fontsize=12)
    IC555 = elm.Ic555(size=(5,8))
    gnd = elm.Ground().at(IC555.GND)
    elm.Dot()
    elm.Resistor().endpoints(IC555.DIS, IC555.THR).label('100 kΩ')
    elm.Resistor().up().at(IC555.DIS).label('1 kΩ').label('+Vcc', 'right')
    elm.Line().endpoints(IC555.THR, IC555.TRG)
```

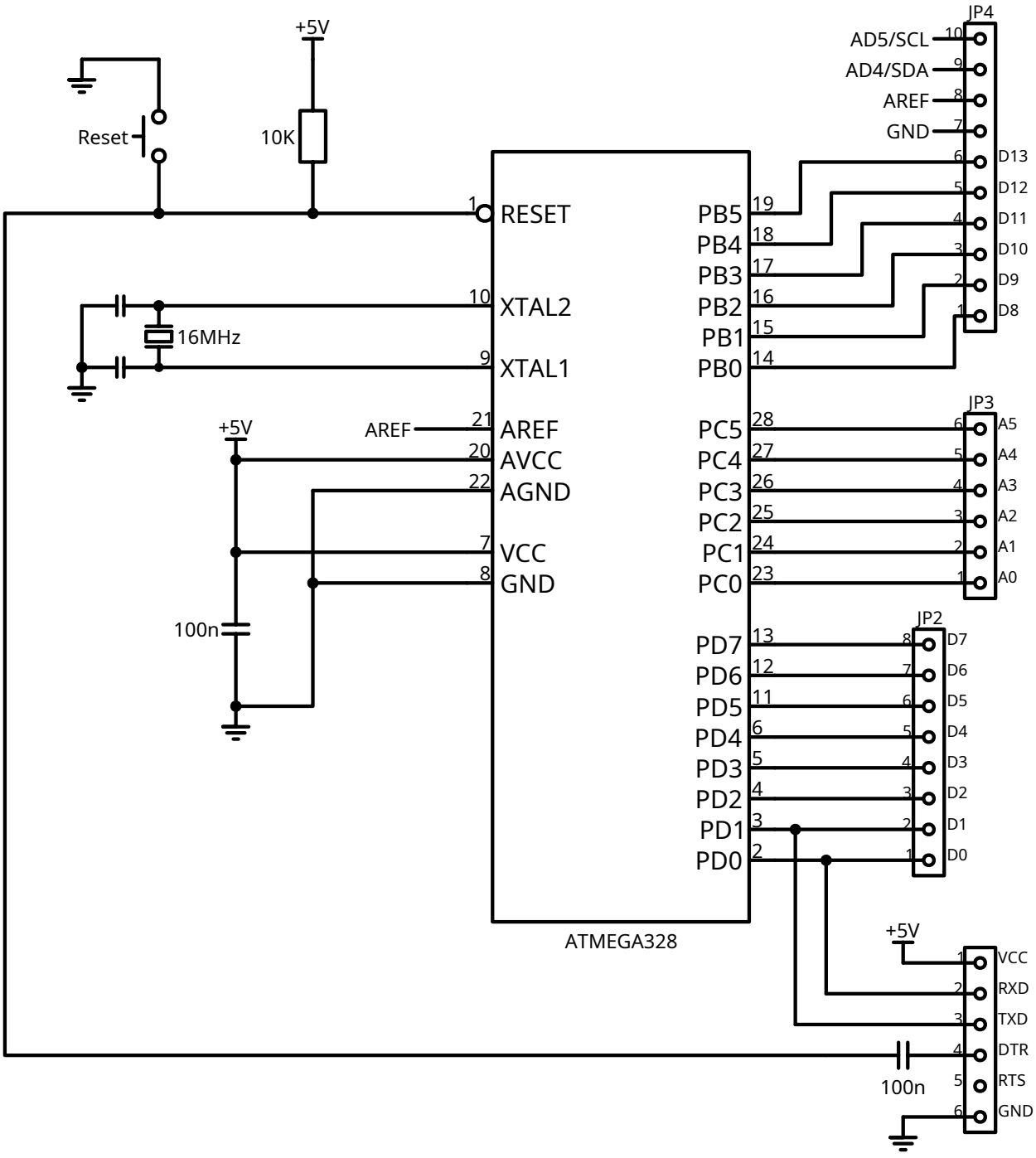
(continues on next page)

(continued from previous page)

```
    elm.Capacitor(polar=True).at(IC555.TRG).toy(gnd.start).label('10 μF')
    elm.Line().tox(gnd.start)
    elm.Capacitor().at(IC555.CTL).toy(gnd.start).label('.01 μF', 'bottom')
    elm.Line().tox(gnd.start)

    elm.Dot().at(IC555.DIS)
    elm.Dot().at(IC555.THR)
    elm.Dot().at(IC555.TRG)
    elm.Line().endpoints(IC555.RST,IC555.Vcc).dot()
    elm.Line().up(d.unit/4).label('+Vcc', 'right')
    d.move_from(IC555.OUT, dx=5, dy=-1)

    IC4026 = (elm.Ic()
              .pin('L', 'CLK', pin='1')
              .pin('L', 'INH', pin='2') # Inhibit
              .pin('L', 'RST', pin='15')
              .pin('L', 'DEI', pin='3') # Display Enable In
              .pin('B', 'Vss', pin='8')
              .pin('T', 'Vdd', pin='16')
              .pin('B', 'UCS', pin='14') # Ungated C Segment
              .pin('B', 'DEO', pin='4')  # Display Enable Out
              .pin('B', 'Co', pin='4')   # Carry out
              .pin('R', 'g', pin='7')
              .pin('R', 'f', pin='6')
              .pin('R', 'e', pin='11')
              .pin('R', 'd', pin='9')
              .pin('R', 'c', pin='13')
              .pin('R', 'b', pin='12')
              .pin('R', 'a', pin='10')
              .side('B', spacing=1, pad=.75, leadlen=.75)
              .side('L', spacing=1, pad=1.5, leadlen=.6)
              .label('4026').right().anchor('center'))
    elm.Wire('c').at(IC555.OUT).to(IC4026.CLK)
    elm.Line().endpoints(IC4026.INH, IC4026.RST).dot()
    elm.Line().left(d.unit/4)
    elm.Ground()
    elm.Wire('|-').at(IC4026.DEI).to(IC4026.Vdd).dot()
    elm.Line().up(d.unit/4).label('+Vcc', 'right')
    elm.Line().at(IC4026.Vss).tox(IC4026.UCS).dot()
    elm.Ground()
    elm.Line().tox(IC4026.DEO).dot()
    elm.Line().tox(IC4026.Co)

    elm.Resistor().right().at(IC4026.a)
    disp = elm.SevenSegment(cathode=True).anchor('a')
    elm.Resistor().at(IC4026.b)
    elm.Resistor().at(IC4026.c)
    elm.Resistor().at(IC4026.d)
    elm.Resistor().at(IC4026.e)
    elm.Resistor().at(IC4026.f)
    elm.Resistor().at(IC4026.g).label('7 x 330', loc='bottom')
    elm.Ground(lead=False).at(disp.cathode)
```

4.6.3 Arduino Board

The Arduino board uses *schemdraw.elements.connectors.OrthoLines* to easily add all connections between data bus and headers.




```
class Atmega328(elm.Ic):
    def __init__(self, *args, **kwargs):
        pins=[elm.IcPin(name='PD0', pin='2', side='r', slot='1/22'),
              elm.IcPin(name='PD1', pin='3', side='r', slot='2/22'),
```

(continues on next page)

(continued from previous page)

```
            elm.IcPin(name='PD2', pin='4', side='r', slot='3/22'),
            elm.IcPin(name='PD3', pin='5', side='r', slot='4/22'),
            elm.IcPin(name='PD4', pin='6', side='r', slot='5/22'),
            elm.IcPin(name='PD5', pin='11', side='r', slot='6/22'),
            elm.IcPin(name='PD6', pin='12', side='r', slot='7/22'),
            elm.IcPin(name='PD7', pin='13', side='r', slot='8/22'),
            elm.IcPin(name='PC0', pin='23', side='r', slot='10/22'),
            elm.IcPin(name='PC1', pin='24', side='r', slot='11/22'),
            elm.IcPin(name='PC2', pin='25', side='r', slot='12/22'),
            elm.IcPin(name='PC3', pin='26', side='r', slot='13/22'),
            elm.IcPin(name='PC4', pin='27', side='r', slot='14/22'),
            elm.IcPin(name='PC5', pin='28', side='r', slot='15/22'),
            elm.IcPin(name='PB0', pin='14', side='r', slot='17/22'),
            elm.IcPin(name='PB1', pin='15', side='r', slot='18/22'),
            elm.IcPin(name='PB2', pin='16', side='r', slot='19/22'),
            elm.IcPin(name='PB3', pin='17', side='r', slot='20/22'),
            elm.IcPin(name='PB4', pin='18', side='r', slot='21/22'),
            elm.IcPin(name='PB5', pin='19', side='r', slot='22/22'),

            elm.IcPin(name='RESET', side='l', slot='22/22', invert=True, pin='1'),
            elm.IcPin(name='XTAL2', side='l', slot='19/22', pin='10'),
            elm.IcPin(name='XTAL1', side='l', slot='17/22', pin='9'),
            elm.IcPin(name='AREF', side='l', slot='15/22', pin='21'),
            elm.IcPin(name='AVCC', side='l', slot='14/22', pin='20'),
            elm.IcPin(name='AGND', side='l', slot='13/22', pin='22'),
            elm.IcPin(name='VCC', side='l', slot='11/22', pin='7'),
            elm.IcPin(name='GND', side='l', slot='10/22', pin='8')]
        super().__init__(pins=pins, size=(5, 15), pinspacing=0.6, plblofst=.05, botlabel=
↪'ATMEGA328', **kwargs)


with schemdraw.Drawing() as d:
    d.config(fontsize=11, inches_per_unit=.4)
    Q1 = Atmega328()
    JP4 = (elm.Header(rows=10, shownumber=True,
                      pinsright=['D8', 'D9', 'D10', 'D11', 'D12', 'D13', '', '', '', ''],
                      pinalignright='center')
                .flip().at(Q1.PB5, dx=4, dy=1).anchor('pin6').label('JP4', fontsize=10))

    JP3 = (elm.Header(rows=6, shownumber=True, pinsright=['A0', 'A1', 'A2', 'A3', 'A4',
↪'A5'], pinalignright='center')
                      .flip().at(Q1.PC5, dx=4).anchor('pin6').label('JP3',␣
↪fontsize=10))

    JP2 = (elm.Header(rows=8, shownumber=True, pinsright=['D0', 'D1', 'D2', 'D3', 'D4',
↪'D5', 'D6', 'D7'],
                      pinalignright='center')).at(Q1.PD7, dx=3).flip().anchor('pin8').
↪label('JP2', fontsize=10)

    elm.OrthoLines(n=6).at(Q1.PB5).to(JP4.pin6)
    elm.OrthoLines(n=6).at(Q1.PC5).to(JP3.pin6)
    elm.OrthoLines(n=8).at(Q1.PD7).to(JP2.pin8)
```

(continues on next page)

(continued from previous page)

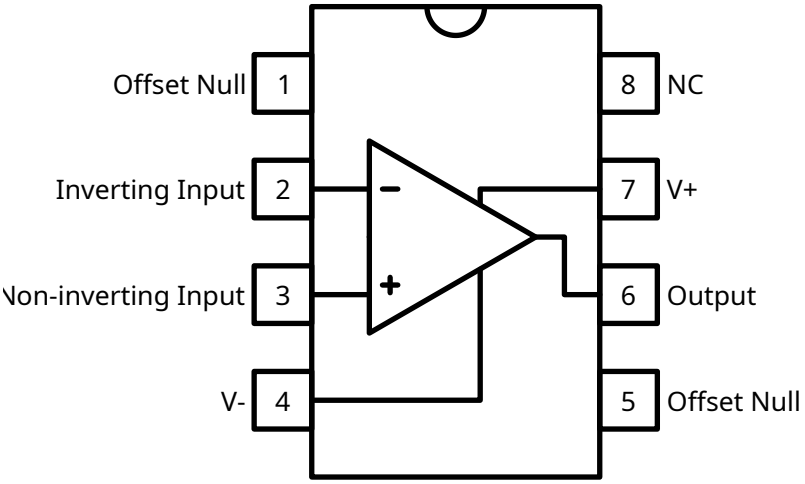
```
    elm.Line().left(.9).at(JP4.pin7).label('GND', 'left')
    elm.Line().left(.9).at(JP4.pin8).label('AREF', 'left')
    elm.Line().left(.9).at(JP4.pin9).label('AD4/SDA', 'left')
    elm.Line().left(.9).at(JP4.pin10).label('AD5/SCL', 'left')

    JP1 = (elm.Header(rows=6, shownumber=True, pinsright=['VCC', 'RXD', 'TXD', 'DTR',
↪'RTS', 'GND'],
                      pinalignright='center').right().at(Q1.PD0, dx=4, dy=-2).anchor(
↪'pin1'))
    elm.Line().left(d.unit/2).at(JP1.pin1)
    elm.Vdd().label('+5V')
    elm.Line().left().at(JP1.pin2)
    elm.Line().toy(Q1.PD0).dot()
    elm.Line().left(d.unit+.6).at(JP1.pin3)
    elm.Line().toy(Q1.PD1).dot()
    elm.Line().left(d.unit/2).at(JP1.pin6)
    elm.Ground()

    elm.Line().left(d.unit*2).at(Q1.XTAL2).dot()
    with d.hold():
        elm.Capacitor().left(d.unit/2).scale(.75)
        elm.Line().toy(Q1.XTAL1).dot()
        elm.Ground()
        elm.Capacitor().right(d.unit/2).scale(.75).dot()
    elm.Crystal().toy(Q1.XTAL1).label('16MHz', 'bottom')
    elm.Line().tox(Q1.XTAL1)

    elm.Line().left(d.unit/3).at(Q1.AREF).label('AREF', 'left')
    elm.Line().left(1.5*d.unit).at(Q1.AVCC)
    elm.Vdd().label('+5V')
    elm.Line().toy(Q1.VCC).dot().idot()
    elm.Line().tox(Q1.VCC).hold()
    elm.Capacitor().down().label('100n')
    GND = elm.Ground()

    elm.Line().left().at(Q1.AGND)
    elm.Line().toy(Q1.GND).dot()
    elm.Line().tox(Q1.GND).hold()
    elm.Wire('|-').to(GND.center).dot()

    elm.Line().left().at(Q1.RESET).dot()
    with d.hold():
        elm.RBox().up().label('10K')
        elm.Vdd().label('+5V')
    elm.Line().left().dot()
    with d.hold():
        RST = elm.Button().up().label('Reset')
        elm.Line().left(d.unit/2)
        elm.Ground()

    elm.Capacitor().left().at(JP1.pin4).label('100n', 'bottom')
```

(continues on next page)

(continued from previous page)

```
    elm.Wire('c', k=-16).to(RST.start)
```

4.6.4 741 Opamp, DIP Layout



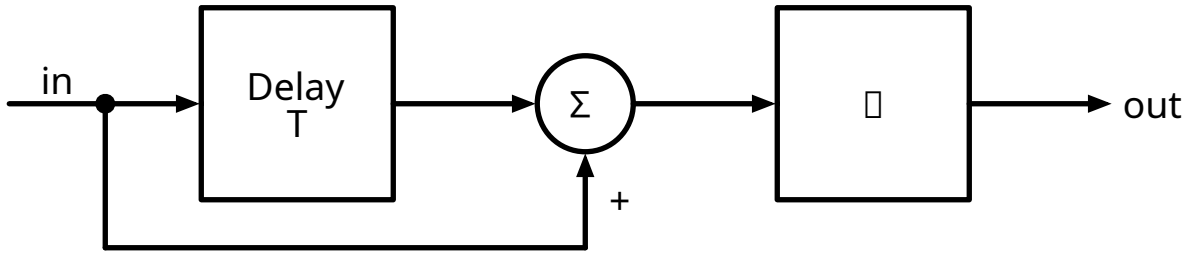

```
with schemdraw.Drawing() as d:
    Q = (elm.IcDIP(pins=8)
                .label('Offset Null', loc='p1', fontsize=10)
                .label('Inverting Input', loc='p2', fontsize=10)
                .label('Non-inverting Input', loc='p3', fontsize=10)
                .label('V-', loc='p4', fontsize=10)
                .label('Offset Null', loc='p5', fontsize=10)
                .label('Output', loc='p6', fontsize=10)
                .label('V+', loc='p7', fontsize=10)
                .label('NC', loc='p8', fontsize=10))
    elm.Line().at(Q.p2_in).length(d.unit/5)
    op = elm.Opamp().anchor('in1').scale(.8)
    elm.Line().at(Q.p3_in).length(d.unit/5)
    elm.Wire('c', k=.3).at(op.out).to(Q.p6_in)
    elm.Wire('-|').at(Q.p4_in).to(op.n1)
    elm.Wire('-|').at(Q.p7_in).to(op.n2)
```

4.7 Signal Processing

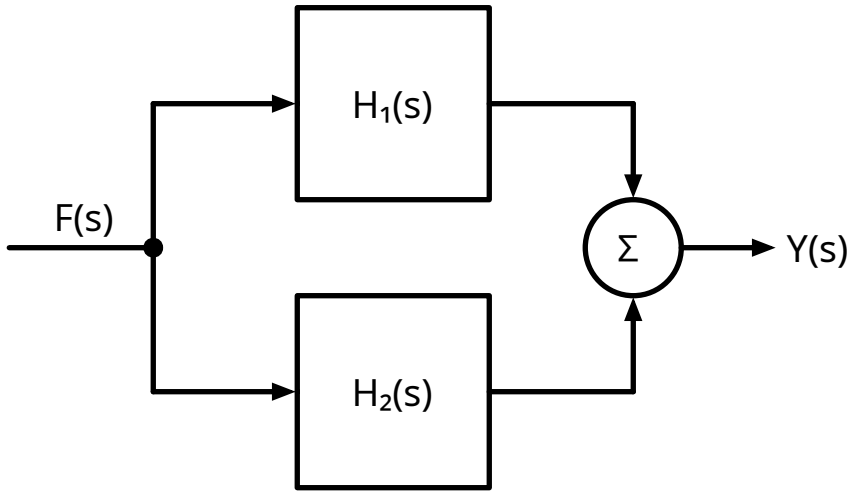
Signal processing elements are in the *schemdraw.dsp.dsp* module.

```
from schemdraw import dsp
```

4.7.1 Various Networks




```
with schemdraw.Drawing() as d:
    dsp.Line().length(d.unit/3).label('in')
    inpt = dsp.Dot()
    dsp.Arrow().length(d.unit/3)
    delay = dsp.Box(w=2, h=2).anchor('W').label('Delay\nT')
    dsp.Arrow().right(d.unit/2).at(delay.E)
    sm = dsp.SumSigma()
    dsp.Arrow().at(sm.E).length(d.unit/2)
    intg = dsp.Box(w=2, h=2).anchor('W').label(r'$\int$')
    dsp.Arrow().right(d.unit/2).at(intg.E).label('out', loc='right')
    dsp.Line().down(d.unit/2).at(inpt.center)
    dsp.Line().tox(sm.S)
    dsp.Arrow().toy(sm.S).label('+', loc='bot')
```




```
with schemdraw.Drawing() as d:
    d.config(fontsize=14)
    dsp.Line().length(d.unit/2).label('F(s)').dot()
    with d.hold():
        dsp.Line().up(d.unit/2)
        dsp.Arrow().right(d.unit/2)
        h1 = dsp.Box(w=2, h=2).anchor('W').label('$H_1(s)$')
    dsp.Line().down(d.unit/2)
    dsp.Arrow().right(d.unit/2)
    h2 = dsp.Box(w=2, h=2).anchor('W').label('$H_2(s)$')
```

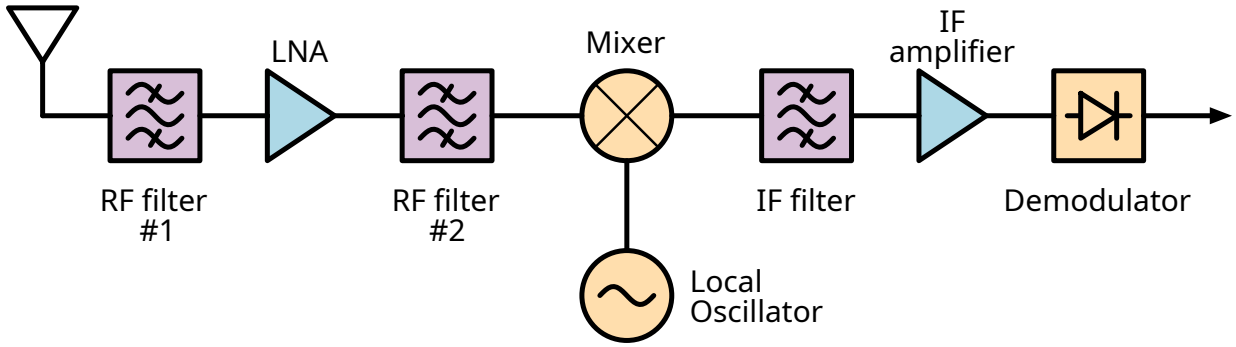
(continues on next page)

(continued from previous page)

```
    sm = dsp.SumSigma().right().at((h1.E[0] + d.unit/2, 0)).anchor('center')
    dsp.Line().at(h1.E).tox(sm.N)
    dsp.Arrow().toy(sm.N)
    dsp.Line().at(h2.E).tox(sm.S)
    dsp.Arrow().toy(sm.S)
    dsp.Arrow().right(d.unit/3).at(sm.E).label('Y(s)', 'right')
```

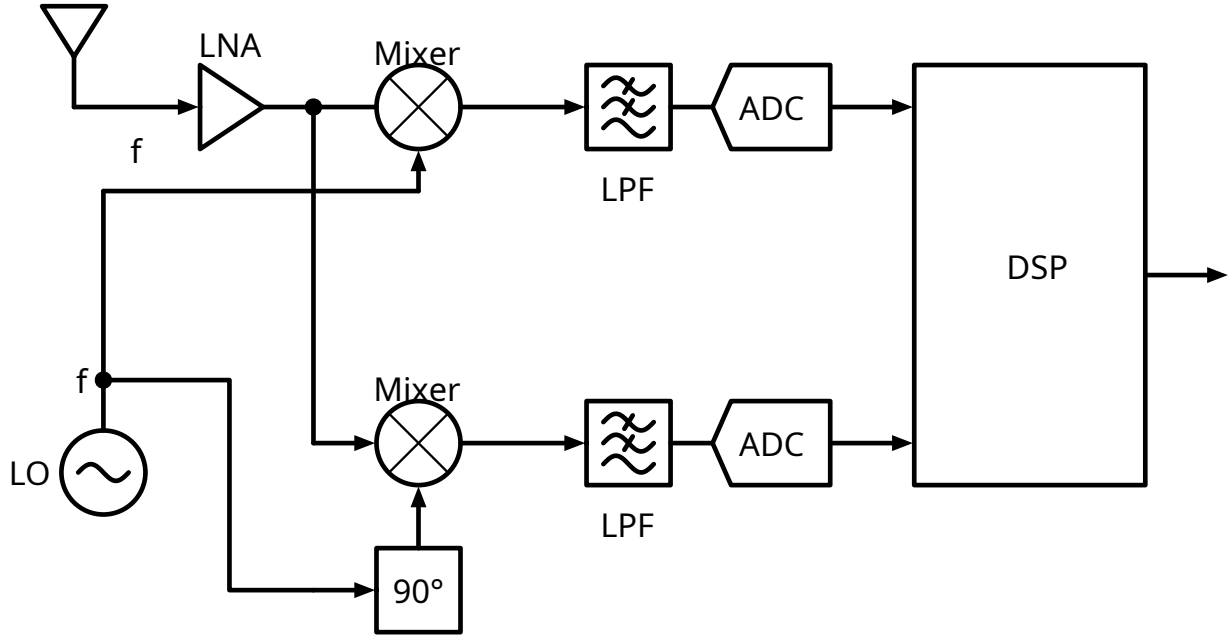
4.7.2 Superheterodyne Receiver

Source.




```
with schemdraw.Drawing() as d:
    d.config(fontsize=12)
    dsp.Antenna()
    dsp.Line().right(d.unit/4)
    dsp.Filter(response='bp').fill('thistle').anchor('W').label('RF filter\n#1', 'bottom
↪', ofst=.2)
    dsp.Line().length(d.unit/4)
    dsp.Amp().fill('lightblue').label('LNA')
    dsp.Line().length(d.unit/4)
    dsp.Filter(response='bp').anchor('W').fill('thistle').label('RF filter\n#2', 'bottom
↪', ofst=.2)
    dsp.Line().length(d.unit/3)
    mix = dsp.Mixer().fill('navajowhite').label('Mixer')
    dsp.Line().at(mix.S).down(d.unit/3)
    dsp.Oscillator().right().anchor('N').fill('navajowhite').label('Local\nOscillator',
↪'right', ofst=.2)
    dsp.Line().at(mix.E).right(d.unit/3)
    dsp.Filter(response='bp').anchor('W').fill('thistle').label('IF filter', 'bottom',␣
↪ofst=.2)
    dsp.Line().right(d.unit/4)
    dsp.Amp().fill('lightblue').label('IF\namplifier')
    dsp.Line().length(d.unit/4)
    dsp.Demod().anchor('W').fill('navajowhite').label('Demodulator', 'bottom', ofst=.2)
    dsp.Arrow().right(d.unit/3)
```

4.7.3 Direct Conversion Receiver



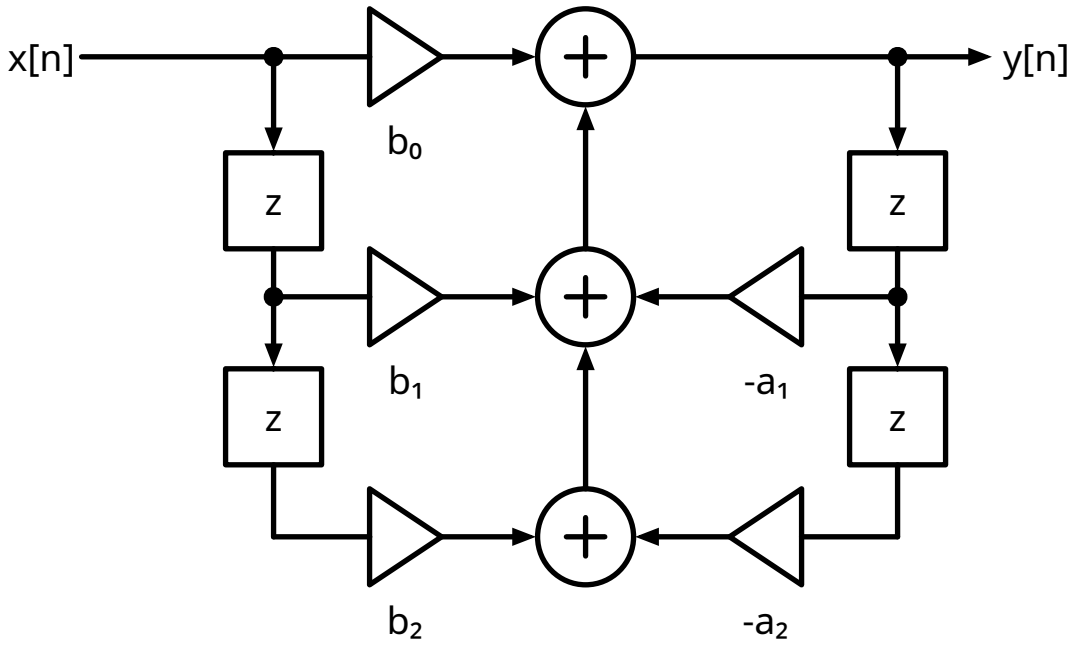

```
with schemdraw.Drawing() as d:
    dsp.Antenna()
    dsp.Arrow().right(d.unit/2).label('$f_{RF}$', 'bot')
    dsp.Amp().label('LNA')
    dsp.Line().right(d.unit/5).dot()
    with d.hold():
        dsp.Line().length(d.unit/4)
        mix1 = dsp.Mixer().label('Mixer', ofst=0)
        dsp.Arrow().length(d.unit/2)
        lpf1 = dsp.Filter(response='lp').label('LPF', 'bot', ofst=.2)
        dsp.Line().length(d.unit/6)
        adc1 = dsp.Adc().label('ADC')
        dsp.Arrow().length(d.unit/3)
        dsp1 = dsp.Ic(pins=[dsp.IcPin(side='L'), dsp.IcPin(side='L'), dsp.IcPin(side='R
↪')],
                      size=(2.75, 5), leadlen=0).anchor('inL2').label('DSP')
        dsp.Arrow().at(dsp1.inR1).length(d.unit/3)

    dsp.Line().toy(dsp1.inL1)
    dsp.Arrow().tox(mix1.W)
    mix2 = dsp.Mixer().label('Mixer', ofst=0)
    dsp.Arrow().tox(lpf1.W)
    dsp.Filter(response='lp').label('LPF', 'bot', ofst=.2)
    dsp.Line().tox(adc1.W)
    dsp.Adc().label('ADC')
    dsp.Arrow().to(dsp1.inL1)

    dsp.Arrow().down(d.unit/6).reverse().at(mix1.S)
    dsp.Line().left(d.unit*1.25)
    dsp.Line().down(d.unit*.75)
```

(continues on next page)

(continued from previous page)

```
    flo = dsp.Dot().label('$f_{LO}$', 'left')
    with d.hold():
        dsp.Line().down(d.unit/5)
        dsp.Oscillator().right().anchor('N').label('LO', 'left', ofst=.15)
    dsp.Arrow().down(d.unit/4).reverse().at(mix2.S)
    b1 = dsp.Square().right().label('90°').anchor('N')
    dsp.Arrow().left(d.unit/4).reverse().at(b1.W)
    dsp.Wire('c', k=-1).to(flo.center)
```

4.7.4 Digital Filter




```
with schemdraw.Drawing() as d:
    d.config(unit=1, fontsize=14)
    dsp.Line().length(d.unit*2).label('x[n]', 'left').dot()

    with d.hold():
        dsp.Line().right()
        dsp.Amp().label('$b_0$', 'bottom')
        dsp.Arrow()
        s0 = dsp.Sum().anchor('W')

    dsp.Arrow().down()
    z1 = dsp.Square(label='$z^{-1}$')
    dsp.Line().length(d.unit/2).dot()

    with d.hold():
        dsp.Line().right()
        dsp.Amp().label('$b_1$', 'bottom')
        dsp.Arrow()
```

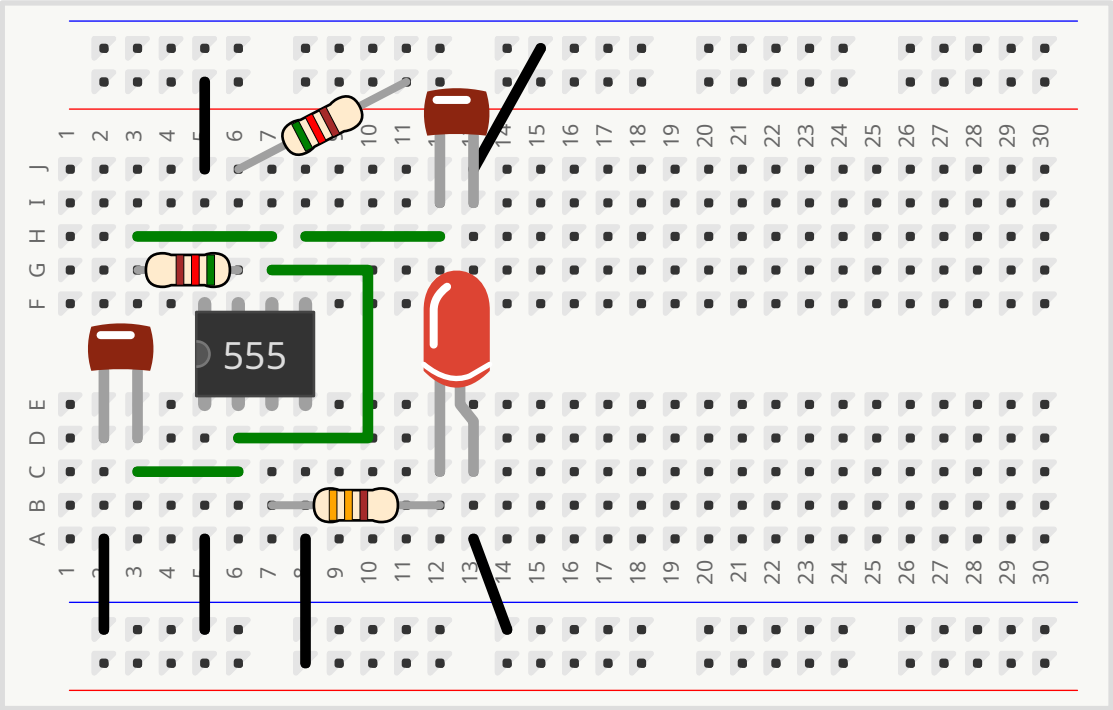
(continues on next page)

(continued from previous page)

```
        s1 = dsp.Sum().anchor('W')

    dsp.Arrow().down(d.unit*.75)
    dsp.Square().label('$z^{-1}$')
    dsp.Line().length(d.unit*.75)
    dsp.Line().right()
    dsp.Amp().label('$b_2$', 'bottom')
    dsp.Arrow()
    s2 = dsp.Sum().anchor('W')

    dsp.Arrow().at(s2.N).toy(s1.S)
    dsp.Arrow().at(s1.N).toy(s0.S)

    dsp.Line().right(d.unit*2.75).at(s0.E).dot()
    dsp.Arrow().right().label('y[n]', 'right').hold()
    dsp.Arrow().down()
    dsp.Square().label('$z^{-1}$')
    dsp.Line().length(d.unit/2).dot()
    with d.hold():
        dsp.Line().left()
        a1 = dsp.Amp().label('$-a_1$', 'bottom')
        dsp.Arrow().at(a1.out).tox(s1.E)

    dsp.Arrow().down(d.unit*.75)
    dsp.Square().label('$z^{-1}$')
    dsp.Line().length(d.unit*.75)
    dsp.Line().left()
    a2 = dsp.Amp().label('$-a_2$', 'bottom')
    dsp.Arrow().at(a2.out).tox(s2.E)
```

4.8 Pictorial Schematics

4.8.1 LED Blinker




```
elm.Line.defaults['lw'] = 4

with schemdraw.Drawing():
    bb = pictorial.Breadboard().up()
    pictorial.DIP().up().at(bb.E5).label('555', color='#DDD')
    elm.Line().at(bb.A8).to(bb.L1_7)
    elm.Line().at(bb.J5).to(bb.R1_4)
    elm.Line().at(bb.A5).to(bb.L2_4).color('black')
    pictorial.Resistor(330).at(bb.B7).to(bb.B12)
    pictorial.LED(lead_length=.3*pictorial.INCH).at(bb.C12)
    elm.Line().at(bb.A13).to(bb.L2_13).color('black')
    pictorial.Resistor(520).at(bb.G6).to(bb.G3)
    pictorial.Resistor(520).at(bb.J6).to(bb.R1_10)
    elm.Line().at(bb.H3).to(bb.H7).color('green')
    elm.Wire('c').at(bb.G7).to(bb.D6).linewidth(4).color('green')
    elm.Line().at(bb.H8).to(bb.H12).color('green')
    elm.Line().at(bb.J13).to(bb.R2_14).color('black')
    pictorial.CapacitorMylar(lead_length=.2*pictorial.INCH).at(bb.I12)
    elm.Line().at(bb.C6).to(bb.C3).color('green')
    pictorial.CapacitorMylar(lead_length=.2*pictorial.INCH).at(bb.D2)
    elm.Line().at(bb.A2).to(bb.L2_1).color('black')
```

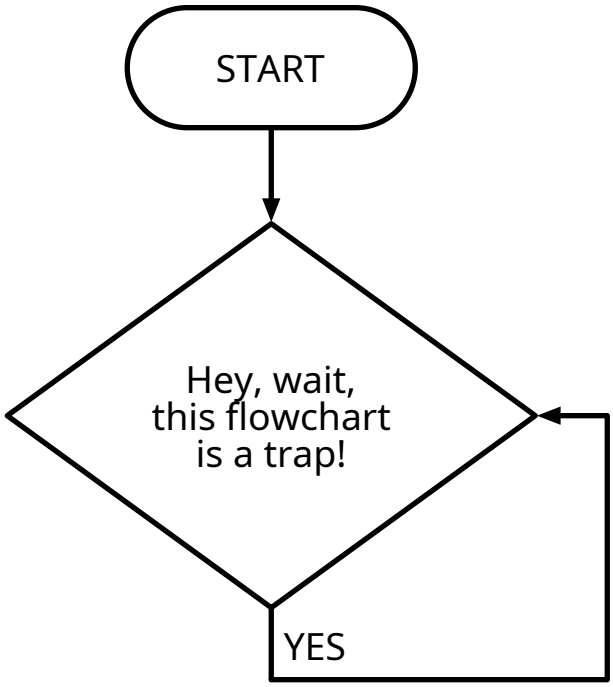
4.9 Flowcharting

Flowchart elements are defined in the `flow` module.

```
from schemdraw import flow
```

4.9.1 It's a Trap!

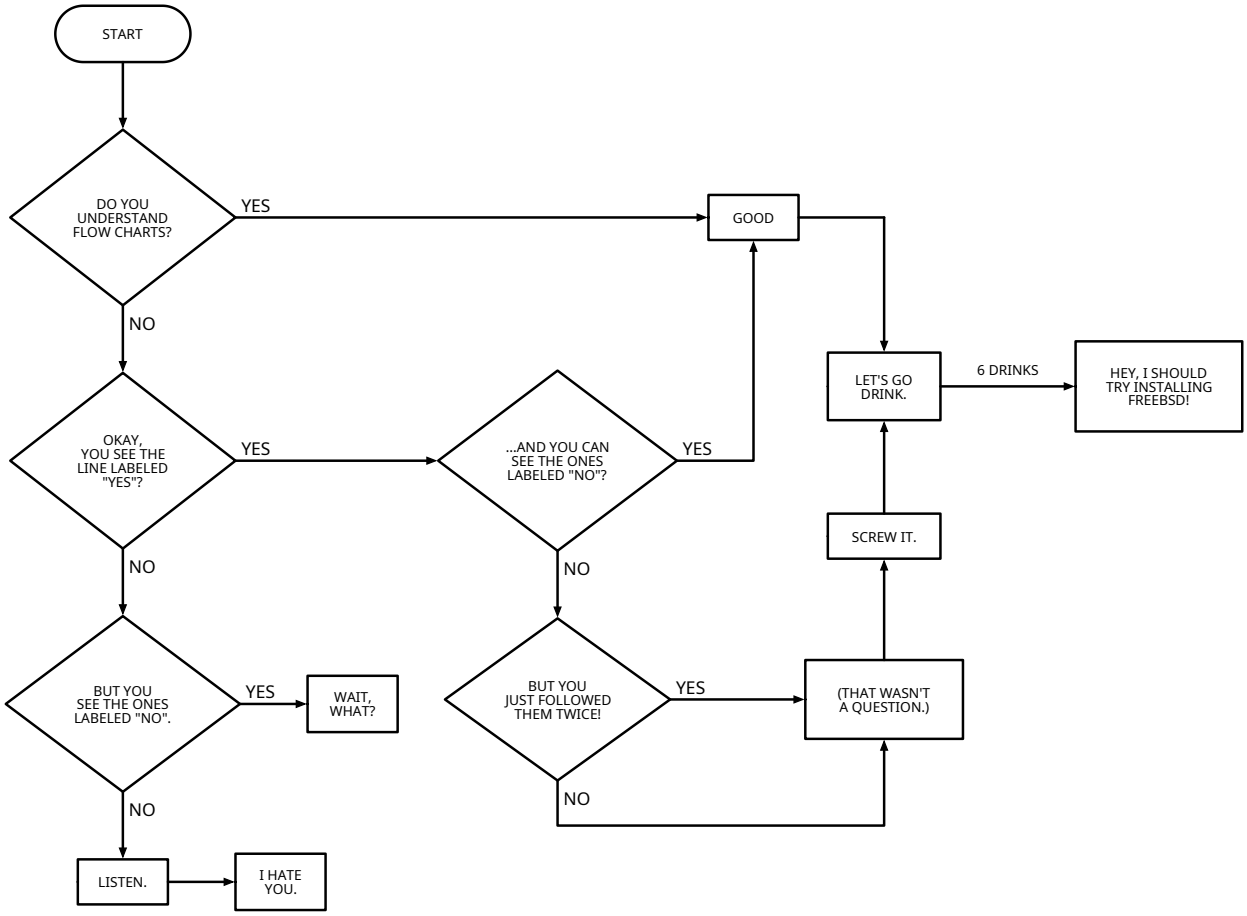
Recreation of XKCD 1195.




```
with schemdraw.Drawing() as d:
    flow.Start().label('START')
    flow.Arrow().down(d.unit/3)
    h = flow.Decision(w=5.5, h=4, S='YES').label('Hey, wait,\nthis flowchart\nis a trap!
↪')
    flow.Line().down(d.unit/4)
    flow.Wire('c', k=3.5, arrow='->').to(h.E)
```

4.9.2 Flowchart for flowcharts

Recreation of XKCD 518.




```
with schemdraw.Drawing() as d:
    d.config(fontsize=11)
    b = flow.Start().label('START')
    flow.Arrow().down(d.unit/2)
    d1 = flow.Decision(w=5, h=3.9, E='YES', S='NO').label('DO YOU\nUNDERSTAND\nFLOW␣
↪CHARTS?')
    flow.Arrow().length(d.unit/2)
    d2 = flow.Decision(w=5, h=3.9, E='YES', S='NO').label('OKAY,\nYOU SEE THE\nLINE␣
↪LABELED\n"YES"?')
    flow.Arrow().length(d.unit/2)
    d3 = flow.Decision(w=5.2, h=3.9, E='YES', S='NO').label('BUT YOU\nSEE THE ONES\
↪nLABELED "NO".')

    flow.Arrow().right(d.unit/2).at(d3.E)
    flow.Box(w=2, h=1.25).anchor('W').label('WAIT,\nWHAT?')
    flow.Arrow().down(d.unit/2).at(d3.S)
    listen = flow.Box(w=2, h=1).label('LISTEN.')
    flow.Arrow().right(d.unit/2).at(listen.E)
    hate = flow.Box(w=2, h=1.25).anchor('W').label('I HATE\nYOU.')

    flow.Arrow().right(d.unit*3.5).at(d1.E)
    good = flow.Box(w=2, h=1).anchor('W').label('GOOD')
    flow.Arrow().right(d.unit*1.5).at(d2.E)
```

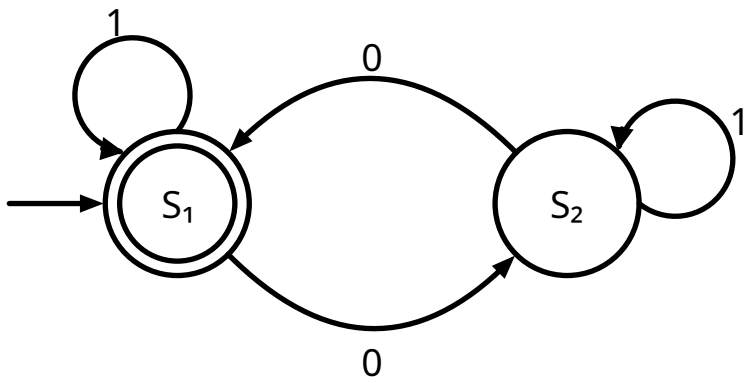
(continues on next page)

(continued from previous page)

```
    d4 = flow.Decision(w=5.3, h=4.0, E='YES', S='NO').anchor('W').label('...AND YOU CAN\
↪nSEE THE ONES\nLABELED "NO"?')

    flow.Wire('-|', arrow='->').at(d4.E).to(good.S)
    flow.Arrow().down(d.unit/2).at(d4.S)
    d5 = flow.Decision(w=5, h=3.6, E='YES', S='NO').label('BUT YOU\nJUST FOLLOWED\nTHEM␣
↪TWICE!')
    flow.Arrow().right().at(d5.E)
    question = flow.Box(w=3.5, h=1.75).anchor('W').label("(THAT WASN'T\nA QUESTION.)")
    flow.Wire('n', k=-1, arrow='->').at(d5.S).to(question.S)

    flow.Line().at(good.E).tox(question.S)
    flow.Arrow().down()
    drink = flow.Box(w=2.5, h=1.5).label("LET'S GO\nDRINK.")
    flow.Arrow().right().at(drink.E).label('6 DRINKS')
    flow.Box(w=3.7, h=2).anchor('W').label('HEY, I SHOULD\nTRY INSTALLING\nFREEBSD!')
    flow.Arrow().up(d.unit*.75).at(question.N)
    screw = flow.Box(w=2.5, h=1).anchor('S').label('SCREW IT.')
    flow.Arrow().at(screw.N).toy(drink.S)
```

4.9.3 State Machine Acceptor

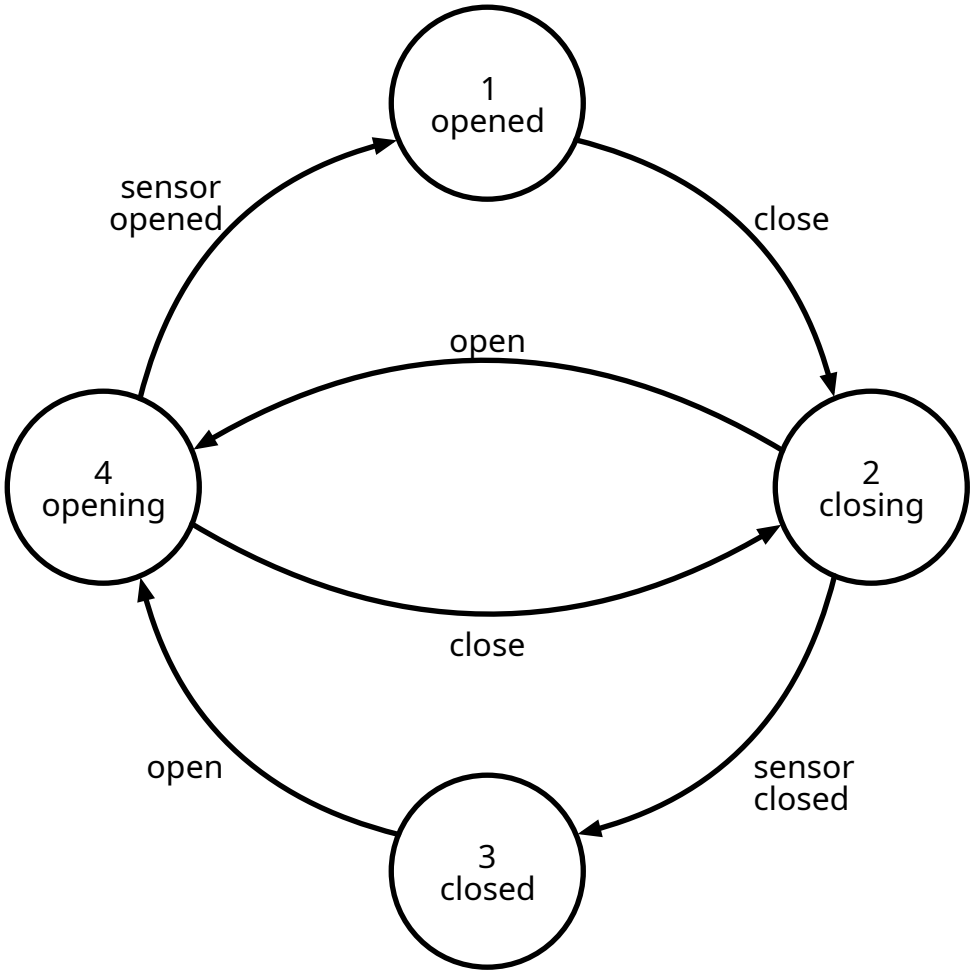
Source




```
with schemdraw.Drawing() as d:
    elm.Arrow().length(1)
    s1 = flow.StateEnd().anchor('W').label('$S_1$')
    elm.Arc2(arrow='<-').at(s1.NE).label('0')
    s2 = flow.State().anchor('NW').label('$S_2$')
    elm.Arc2(arrow='<-').at(s2.SW).to(s1.SE).label('0')
    elm.ArcLoop(arrow='<-').at(s2.NE).to(s2.E).label('1')
    elm.ArcLoop(arrow='<-').at(s1.NW).to(s1.N).label('1')
```

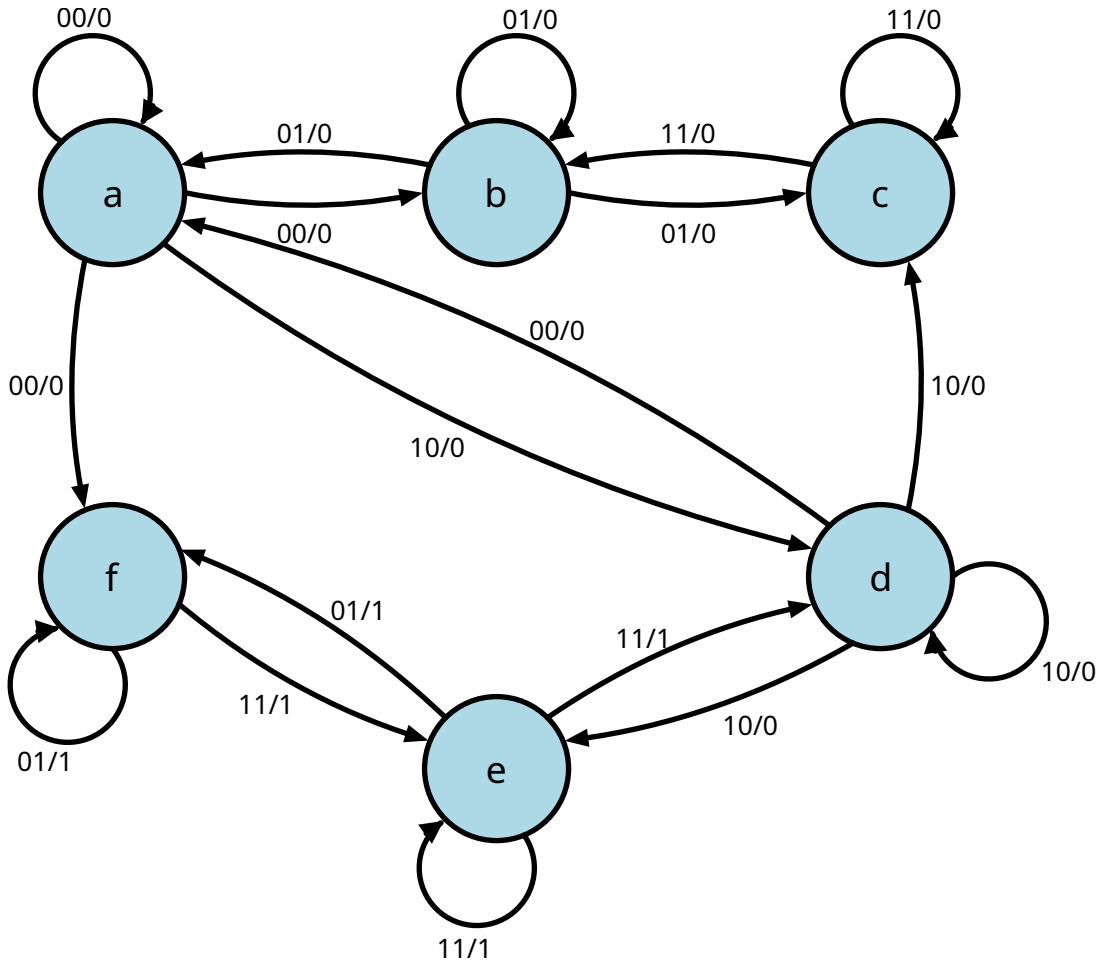
4.9.4 Door Controller

Diagram Source




```
with schemdraw.Drawing() as d:
    d.config(fontsize=12)
    delta = 4
    c4 = flow.Circle(r=1).label('4\nopening')
    c1 = flow.Circle(r=1).at((delta, delta)).label('1\nopened')
    c2 = flow.Circle(r=1).at((2*delta, 0)).label('2\nclosing')
    c3 = flow.Circle(r=1).at((delta, -delta)).label('3\nclosed')
    elm.Arc2(arrow='->', k=.3).at(c4.NNE).to(c1.WSW).label('sensor\nopened')
    elm.Arc2(arrow='->', k=.3).at(c1.ESE).to(c2.NNW).label('close')
    elm.Arc2(arrow='->', k=.3).at(c2.SSW).to(c3.ENE).label('sensor\nclosed')
    elm.Arc2(arrow='->', k=.3).at(c3.WNW).to(c4.SSE).label('open')
    elm.Arc2(arrow='<-', k=.3).at(c4.ENE).to(c2.WNW).label('open')
    elm.Arc2(arrow='<-', k=.3).at(c2.WSW).to(c4.ESE).label('close')
```

4.9.5 Another State Machine



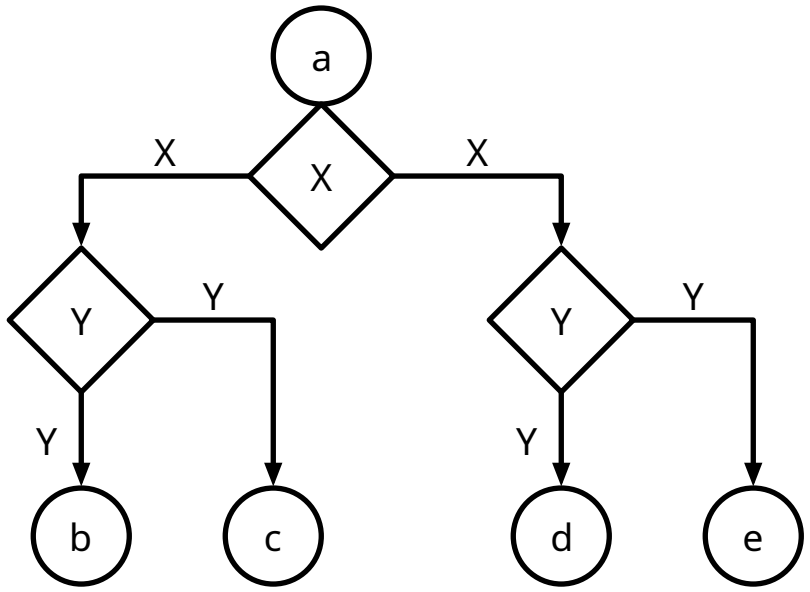

```
with schemdraw.Drawing():
    a = flow.Circle().label('a').fill('lightblue')
    b = flow.Circle().at((4, 0)).label('b').fill('lightblue')
    c = flow.Circle().at((8, 0)).label('c').fill('lightblue')
    f = flow.Circle().at((0, -4)).label('f').fill('lightblue')
    e = flow.Circle().at((4, -6)).label('e').fill('lightblue')
    d = flow.Circle().at((8, -4)).label('d').fill('lightblue')
    elm.ArcLoop(arrow='->').at(a.NW).to(a.NNE).label('00/0', fontsize=10)
    elm.ArcLoop(arrow='->').at(b.NNW).to(b.NE).label('01/0', fontsize=10)
    elm.ArcLoop(arrow='->').at(c.NNW).to(c.NE).label('11/0', fontsize=10)
    elm.ArcLoop(arrow='->').at(d.E).to(d.SE).label('10/0', fontsize=10)
    elm.ArcLoop(arrow='->').at(e.SSE).to(e.SW).label('11/1', fontsize=10)
    elm.ArcLoop(arrow='->').at(f.S).to(f.SW).label('01/1', fontsize=10)
    elm.Arc2(k=.1, arrow='<-').at(a.ENE).to(b.WNW).label('01/0', fontsize=10)
    elm.Arc2(k=.1, arrow='<-').at(b.W).to(a.E).label('00/0', fontsize=10)
    elm.Arc2(k=.1, arrow='<-').at(b.ENE).to(c.WNW).label('11/0', fontsize=10)
    elm.Arc2(k=.1, arrow='<-').at(c.W).to(b.E).label('01/0', fontsize=10)
    elm.Arc2(k=.1, arrow='<-').at(a.ESE).to(d.NW).label('00/0', fontsize=10)
    elm.Arc2(k=.1, arrow='<-').at(d.WNW).to(a.SE).label('10/0', fontsize=10)
```

(continues on next page)

(continued from previous page)

```
    elm.Arc2(k=.1, arrow='<-').at(f.ENE).to(e.NW).label('01/1', fontsize=10)
    elm.Arc2(k=.1, arrow='<-').at(e.WNW).to(f.ESE).label('11/1', fontsize=10)
    elm.Arc2(k=.1, arrow='->').at(e.NE).to(d.WSW).label('11/1', fontsize=10)
    elm.Arc2(k=.1, arrow='->').at(d.SSW).to(e.ENE).label('10/0', fontsize=10)
    elm.Arc2(k=.1, arrow='<-').at(f.NNW).to(a.SSW).label('00/0', fontsize=10)
    elm.Arc2(k=.1, arrow='<-').at(c.SSE).to(d.NNE).label('10/0', fontsize=10)
```

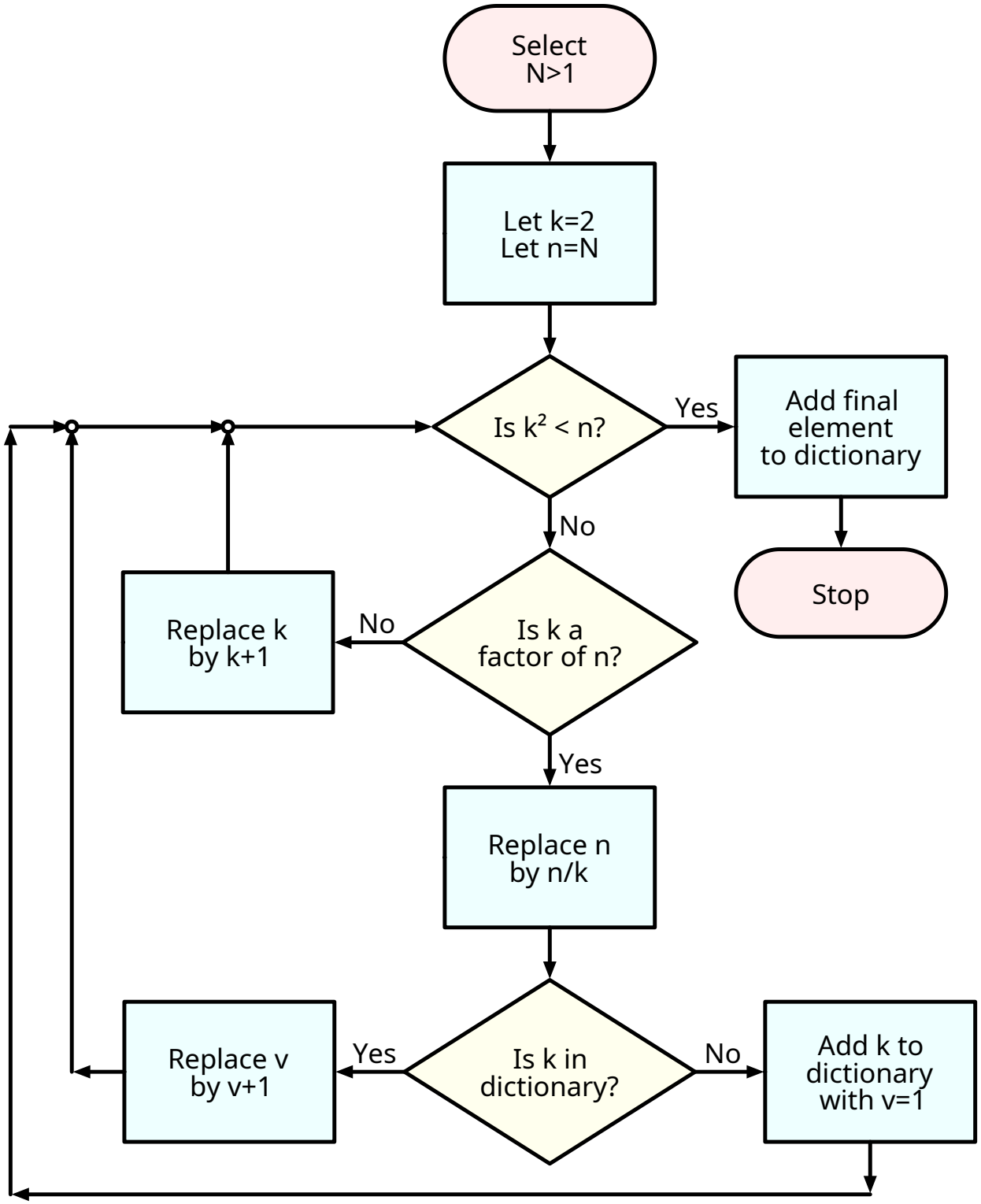
4.9.6 Logical Flow Diagram




```
with schemdraw.Drawing(unit=1) as dwg:
    a = flow.Circle(r=.5).label('a')
    x = flow.Decision(w=1.5, h=1.5).label('$X$').at(a.S).anchor('N')
    elm.RightLines(arrow='->').at(x.E).label(r'$\overline{X}$')
    y1 = flow.Decision(w=1.5, h=1.5).label('$Y$')
    dwg.move_from(y1.N, dx=-5)
    y2 = flow.Decision(w=1.5, h=1.5).label('$Y$')
    elm.RightLines(arrow='->').at(x.W).to(y2.N).label('$X$')
    elm.Arrow().at(y2.S).label('$Y$')
    b = flow.Circle(r=.5).label('b')
    dwg.move_from(b.N, dx=2)
    c = flow.Circle(r=.5).label('c')
    elm.RightLines(arrow='->').at(y2.E).to(c.N).label(r'$\overline{Y}$')
    elm.Arrow().at(y1.S).label('$Y$')
    d = flow.Circle(r=.5).label('d')
    dwg.move_from(d.N, dx=2)
    e = flow.Circle(r=.5).label('e')
    elm.RightLines(arrow='->').at(y1.E).to(e.N).label(r'$\overline{Y}$')
```

4.9.7 Prime Factorization

Chart Source



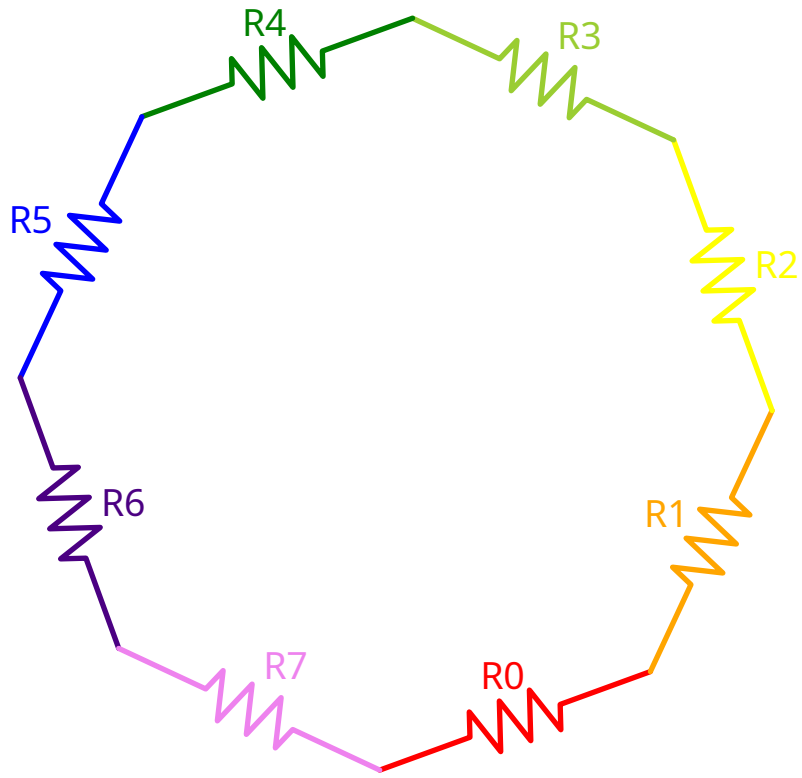
```
# Set default flowchart box fill colors
flow.Box.defaults['fill'] = '#eeffff'
flow.Start.defaults['fill'] = '#ffeeee'
flow.Decision.defaults['fill'] = '#ffffee'

with schemdraw.Drawing() as d:
    d.config(unit=.75)
    flow.Start(h=1.5).label('Select\n$N>1$').drop('S')
    flow.Arrow().down()
    flow.Box().label('Let k=2\nLet $n=N$')
    flow.Arrow()
    k2 = flow.Decision(E='Yes', S='No').label('Is $k^2 < n$?').drop('E')
    flow.Arrow().length(1)
    flow.Box().label('Add final\nelement\nto dictionary').drop('S')
    flow.Arrow().down()
    flow.Start().label('Stop')
    flow.Arrow().at(k2.S)
    kn = flow.Decision(W='No', S='Yes').label('Is $k$ a\nfactor of $n$?').drop('W')
    flow.Arrow().left(1)
    flow.Box().label('Replace $k$\nby $k+1$').drop('N')
    flow.Arrow().toy(k2.W).dot(open=True)
    flow.Arrow().tox(k2.W)

    flow.Arrow().down().at(kn.S)
    flow.Box().label('Replace $n$\nby $n/k$')
    flow.Arrow()
    k3 = flow.Decision(E='No', W='Yes').label('Is $k$ in\ndictionary?').drop('E')

    flow.Arrow().left(1).at(k3.W)
    rep = flow.Box().label('Replace $v$\nby $v+1$')
    flow.Arrow()
    dot = flow.Arrow().up().toy(k2.W).dot(open=True)
    flow.Arrow().right().tox(rep.N)

    flow.Arrow().at(k3.E).right(1)
    flow.Box().label('Add $k$ to\ndictionary\nwith $v=1$').drop('S')
    flow.Arrow().down()
    flow.Arrow().left().to(rep.W, dx=-1.5)
    flow.Arrow().up().toy(k2.W)
    flow.Arrow().right().tox(dot.center)
```

4.10 Styles

Circuit elements can be styled using Matplotlib colors, line-styles, and line widths.

4.10.1 Resistor circle

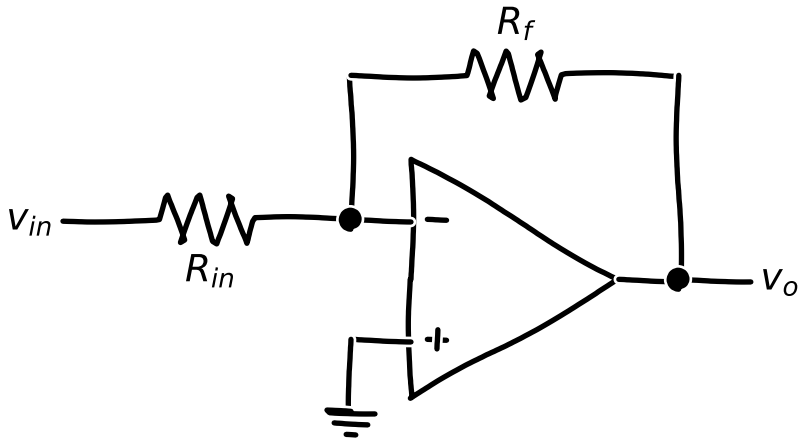
Uses named colors in a loop.




```
with schemdraw.Drawing() as d:
    for i, color in enumerate(['red', 'orange', 'yellow', 'yellowgreen', 'green', 'blue',
↪ 'indigo', 'violet']):
        elm.Resistor().theta(45*i+20).color(color).label('R{}'.format(i))
```

4.10.2 Hand-drawn

And for a change of pace, activate Matplotlib's XKCD mode for "hand-drawn" look!




```
import matplotlib.pyplot as plt
plt.xkcd()
schemdraw.use('matplotlib')

with schemdraw.Drawing() as d:
    op = elm.Opamp(leads=True)
    elm.Line().down().at(op.in2).length(d.unit/4)
    elm.Ground(lead=False)
    Rin = elm.Resistor().at(op.in1).left().idot().label('$R_{in}$', loc='bot').label('$v_
↪{in}$', loc='left')
    elm.Line().up().at(op.in1).length(d.unit/2)
    elm.Resistor().tox(op.out).label('$R_f$')
    elm.Line().toy(op.out).dot()
    elm.Line().right().at(op.out).length(d.unit/4).label('$v_{o}$', loc='right')
```

Need more circuit examples? Check out the Schemdraw Examples Pack on buymeacoffee.com:

CHAPTER

FIVE

CUSTOMIZING ELEMENTS

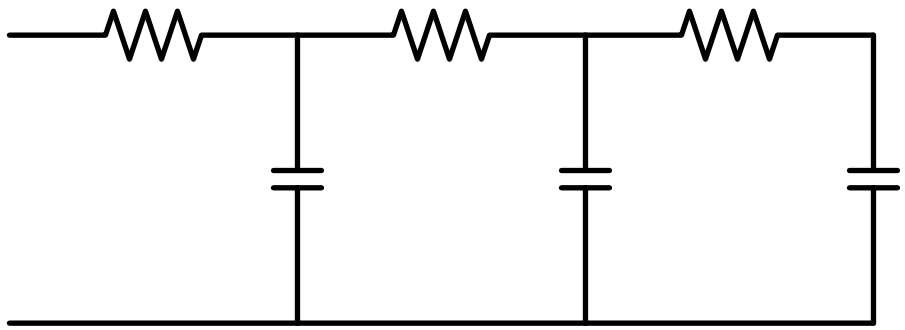
5.1 Grouping Elements

There are two ways to combine multiple elements into a single reusable element. *schemdraw.elements.ElementDrawing* transforms another Drawing into a single element. *schemdraw.elements.compound.ElementCompound* creates a new element from multiple elements.

5.1.1 ElementDrawing

If a set of circuit elements are to be reused multiple times, they can be grouped into a single element. Create and populate a drawing, but set *show=False*. Instead, use the Drawing to create a new *schemdraw.elements.ElementDrawing*, which converts the drawing into an element instance to add to other drawings.

```
with schemdraw.Drawing(show=False) as d1:
    d1 += elm.Resistor()
    d1.push()
    d1 += elm.Capacitor().down()
    d1 += elm.Line().left()
    d1.pop()

with schemdraw.Drawing() as d2:  # Add a second drawing
    for i in range(3):
        d2 += elm.ElementDrawing(d1)   # Add the first drawing to it 3 times
```



5.1.2 ElementCompound

An *schemdraw.elements.compound.ElementCompound* acts like a Drawing on which other elements may be added. Elements are added in the *setup* method.

```
class RC(elm.ElementCompound):
    def setup(self):
        self.r = self.add(elm.Resistor().length(1).scale(.5))
        self.c = self.add(elm.Capacitor().length(1).scale(.5))
        self.elmparams['drop'] = self.c.end

with schemdraw.Drawing():
    r1 = RC()
    r2 = RC()
```

Elements assigned to an attribute of the class will have absolute anchor positions accessible via the index. For example, to get the ending position of the resistor in the second RC:

```
r2['r.end']
```

```
Point(3.0,0.0)
```

5.2 Defining custom elements

All elements are subclasses of *schemdraw.elements.Element* or *schemdraw.elements.Element2Term*. Subclasses only need to define the *__init__* method in order to add lines, shapes, and text to the new element, all of which are defined using *schemdraw.segments.Segment* classes. New Segments should be appended to the *Element.segments* attribute list.

Coordinates are all defined in element cooridnates, where the element begins at (0, 0) and is drawn from left to right. The drawing engine will rotate and translate the element to its final position, and for two-terminal elements deriving from Element2Term, will add lead extensions to the correct length depending on the element's placement parameters. Therefore elements deriving from Element2Term should not define the lead extensions (e.g. a Resistor only defines the zig-zag portion). A standard resistor is 1 drawing unit long, and with default lead extension will become 3 units long.

Segments include *schemdraw.segments.Segment*, *schemdraw.segments.SegmentPoly*, *schemdraw.segments.SegmentCircle*, *schemdraw.segments.SegmentArc*, *schemdraw.segments.SegmentText*, and *schemdraw.segments.SegmentBezier*.

As an example, here's the definition of our favorite element, the resistor:

```
class Resistor(Element2Term):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.segments.append(Segment([(0, 0),
                                      (0.5*reswidth, resheight),
                                      (1.5*reswidth, -resheight),
                                      (2.5*reswidth, resheight),
                                      (3.5*reswidth, -resheight),
                                      (4.5*reswidth, resheight),
```

(continues on next page)

(continued from previous page)

```
                                        (5.5*reswidth, -resheight),
                                        (6*reswidth, 0)]))
```

The resistor is made of one path. *reswidth* and *resheight* are constants that define the height and width of the resistor zigzag (and are referenced by several other elements too). Browse the source code in the *Schemdraw.elements* submodule to see the definitions of the other built-in elements.

In addition to the list of Segments, any named anchors and other parameters should be specified. Anchors should be added to the *Element.anchors* dictionary as {name: (x, y)} key/value pairs.

5.2.1 Parameters and Defaults

Element subclasses may have an *_element_defaults* class attribute dictionary to specify default parameters used for drawing the element. This dictionary will be ChainMapped with the *_element_defaults* from all its parent classes into the *Element.defaults* dictionary the user may change to modify default behaviors.

To access any of these parameters when defining the element, use the *self.params* dictionary, which ensures the correct parameter, whether a default value, a default from a parent class, or a parameter overriden by the user, is obtained. Any non-*None* named arguments provided to the Element will be inserted into *self.params* automatically (by the *Element.__new__* method).

Parameters that need to be set dynamically during instantiation should be set in the *self.elmparams* dictionary, so they may still be overriden by the user.

For example, consider the *Dot* element:

```
class Dot(Element):
    ''' Connection Dot

        Keyword Args:
            radius: Radius of dot [default: 0.075]
            open: Draw as an open circle [default: False]
    '''
    _element_defaults = {
        'radius': 0.075,
        'open': False}
    def __init__(self,
                 radius: Optional[float] = None,
                 open: Optional[bool] = None,
                 **kwargs):
        super().__init__(**kwargs)
        fill = 'bg' if self.params['open'] else True
        self.elmparams['fill'] = fill
        self.segments.append(SegmentCircle((0, 0), self.params['radius']))
        self.anchors['center'] = (0, 0)
```

It contains two default parameters, *radius*, and *open*. The user may override these for every new Dot by setting *Dot.defaults['radius'] = value*. Or to override the defaults for a single instance of Dot, provide the parameter at instantiation: *Dot(radius=value)*.

Inside the *Dot.__init__* method, the *fill* parameter is determined based on the value of the *open* parameter, read from *self.params['open']*. The Dot is filled when the dot is not open, but filled with background color ('bg') when the dot is open. Because the fill value was added to *self.elmparams*, the user may sitll specify their own fill color using *Dot(fill=color)*.

Next, a *SegmentCircle* is added with radius taken from *self.params['radius']*, so that the default radius will be used unless overridden. Finally, an anchor named *center* is defined at the center of the dot.

When drawn, the parameters for the element are obtained from a ChainMap of the parameters in this order of preference:

1) Setter methods like *.fill()* or *.color()* called after the element is instantiated
2) Named arguments provided to Element instantiation
3) Defaults set by the user in Element.defaults (inheriting from parent classes)
4) Parameters defined in the Element attribute *self.elmparams*
5) Parameters defined by Drawing.config
6) Parameters defined by Schemdraw.config

5.2.2 Flux Capacitor Example

For an example, let's make a flux capacitor circuit element.

Since everyone knows a flux-capacitor has three branches, we should subclass the standard *schemdraw.elements.Element* class instead of *schemdraw.elements.Element2Term*. Start by importing the Segments and define the class name and *__init__* function:

```
from schemdraw.segments import *

class FluxCapacitor(Element):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
```

The *kwargs* are passed to *super* to initialize the Element.

We want a dot in the center of our flux capacitor, so start by adding a *SegmentCircle*. The *fclen* and *radius* variables could be set as arguments to the __init__ and/or added to *_element_defaults* for the user to adjust, if desired, but here they are defined as constants in the __init__.

```
fclen = 0.5
radius = 0.075
self.segments.append(SegmentCircle((0, 0), radius))
```

Next, add the paths as Segment instances, which are drawn as lines. The flux capacitor will have three paths, all extending from the center dot:

```
self.segments.append(Segment([(0, 0), (0, -fclen*1.41)]))
self.segments.append(Segment([(0, 0), (fclen, fclen)]))
self.segments.append(Segment([(0, 0), (-fclen, fclen)]))
```

And at the end of each path is an open circle. Append three more *SegmentCircle* instances. By specifying *fill=None* the SegmentCircle will always remain unfilled regardless of any *fill* arguments provided to *Drawing* or *FluxCapacitor*.

```
self.segments.append(SegmentCircle((0, -fclen*1.41), 0.2, fill=None))
self.segments.append(SegmentCircle((fclen, fclen), 0.2, fill=None))
self.segments.append(SegmentCircle((-fclen, fclen), 0.2, fill=None))
```

Finally, we need to define anchor points so that other elements can be connected to the right places. Here, they're called *p1*, *p2*, and *p3* for lack of better names (what do you call the inputs to a flux capacitor?) Add these to the *self.anchors* dictionary.

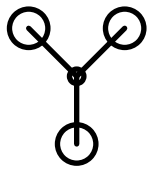
```
self.anchors['p1'] = (-fclen, fclen)
self.anchors['p2'] = (fclen, fclen)
self.anchors['p3'] = (0, -fclen*1.41)
```

Here's the Flux Capacitor class all in one:

```
class FluxCapacitor(elm.Element):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        radius = 0.075
        fclen = 0.5
        self.segments.append(SegmentCircle((0, 0), radius))
        self.segments.append(Segment([(0, 0), (0, -fclen*1.41)]))
        self.segments.append(Segment([(0, 0), (fclen, fclen)]))
        self.segments.append(Segment([(0, 0), (-fclen, fclen)]))
        self.segments.append(SegmentCircle((0, -fclen*1.41), 0.2, fill=None))
        self.segments.append(SegmentCircle((fclen, fclen), 0.2, fill=None))
        self.segments.append(SegmentCircle((-fclen, fclen), 0.2, fill=None))
        self.anchors['p1'] = (-fclen, fclen)
        self.anchors['p2'] = (fclen, fclen)
        self.anchors['p3'] = (0, -fclen*1.41)
```

Try it out:

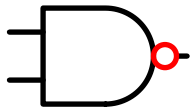
```
FluxCapacitor()
```



5.3 Segment objects

After an element is added to a drawing, the *schemdraw.segments.Segment* objects defining it are accessible in the *segments* attribute list of the Element. For even more control over customizing individual pieces of an element, the parameters of a Segment can be changed.

```
d += (n := logic.Nand())
n.segments[1].color = 'red'
n.segments[1].zorder = 5  # Put the bubble on top
```



5.4 Matplotlib axis

When using the Matplotlib backend (the default), a final customization option is to use the Matplotlib figure and add to it. A `schemdraw.Figure` is returned from the *draw* method, which contains *fig* and *ax* attributes holding the Matplotlib figure.

```
schemdraw.use('matplotlib')
d = schemdraw.Drawing()
d.add(elm.Resistor())
schemfig = d.draw()
schemfig.ax.axvline(.5, color='purple', ls='--')
schemfig.ax.axvline(2.5, color='orange', ls='-', lw=3);
display(schemfig)
```

CHAPTER

SIX

CLASS DEFINITIONS

6.1 Drawing

class schemdraw.**Drawing**(*canvas=None*, *file=None*, *show=True*, *transparent=False*, *dpi=72*, ***kwargs*)

A schematic drawing

See *schemdraw.config* method for argument defaults

Parameters

- **canvas** (Optional[Union[Backends, xml.etree.ElementTree.Element, matplotlib.pyplot.Axes]]) – Canvas to draw on when using Drawing context manager. Can be string 'matplotlib' or 'svg' to create new canvas with these backends, or an instance of a matplotlib axis, or an instance of xml.etree.ElementTree containing SVG. Default is value set by schemdraw.use().
- **file** (Optional[str]) – optional filename to save on exiting context manager or calling draw method.
- **show** (bool) – Show the drawing after exiting context manager
- **transparent** (bool) – Save to file with a transparent background
- **dpi** (float) – Dots-per-inch when saving to a raster graphics file

here

(xy tuple) Current drawing position. The next element will be added at this position unless specified otherwise.

theta

(float) Current drawing angle, in degrees. The next element will be added with this angle unless specified otherwise.

class HoldState(*dwg*)

Context manager to save the drawing state

add(*element*)

Add an element to the drawing.

Parameters
element (*Element*) – The element to add.

Return type
Element

add_elements(**elements*)

Add multiple elements to the drawing

Return type
None

add_svgdef(*svgdef*)

Add an item to the SVG <defs> element

Return type
None

config(*unit=None*, *inches_per_unit=None*, *fontsize=None*, *font=None*, *color=None*, *lw=None*, *ls=None*, *fill=None*, *bgcolor=None*, *margin=None*, *mathfont=None*)

Set Drawing configuration, overriding schemdraw global config.

Parameters

- **unit** (Optional[float]) – Full length of a 2-terminal element. Inner zig-zag portion of a resistor is 1.0 units.
- **inches_per_unit** (Optional[float]) – Inches per drawing unit for setting drawing scale
- **fontsize** (Optional[float]) – Default font size for text labels
- **font** (Optional[str]) – Default font family for text labels
- **mathfont** (Optional[str]) – Default font family or filename math text delimited by $..$
- **color** (Optional[str]) – Default color name or RGB (0-1) tuple
- **lw** (Optional[float]) – Default line width for elements
- **ls** (Optional[Literal['-', ':', '--', '-.']]) – Default line style
- **fill** (Optional[str]) – Deault fill color for closed elements
- **margin** (Optional[float]) – White space around the drawing in drawing units

Return type
None

container(*cornerradius=0.3*, *padx=0.75*, *pady=0.75*)

Add a container to the drawing. Use as a context manager, such that elemnents inside the *with* are surrounded by the container.

```
>>> with drawing.container():
>>>     elm.Resistor()
>>>     ...
```

Parameters

- **cornerradius** (float) – radius for box corners
- **padx** (float) – space between contents and border in x direction
- **pady** (float) – space between contents and border in y direction

draw(*show=True*, *canvas=None*)

Draw the schematic

Parameters

- **show** (bool) – Show the schematic in a GUI popup window (when outside of a Jupyter inline environment)
- **canvas** – 'matplotlib', 'svg', or Axis instance to draw on

Returns
schemdraw Figure object

`get_bbox()`

Get drawing bounding box

Return type
BBox

`get_imagedata(fmt='svg')`

Get image data as bytes array

Parameters
fmt (ImageFormat | ImageType) – Format or file extension of the image type. SVG backend only supports 'svg' format.

Return type
bytes

Returns
Image data as bytes

`get_segments()`

Get flattened list of all segments in the drawing

Return type
list[Union[*Segment*, *SegmentText*, *SegmentPoly*, *SegmentArc*, *SegmentCircle*, *SegmentBezier*, *SegmentPath*, *SegmentImage*]]

`hold()`

Return context manager to hold the current drawing state and restore it when the with-block ends

`interactive(interactive=True)`

Enable interactive mode (matplotlib backend only). Matplotlib must also be set to interactive with *plt.ion()*.

`move(dx=0, dy=0)`

Move the current drawing position

Parameters

- **dx** (float) – change in x position
- **dy** (float) – change in y position

Return type
None

`move_from(ref, dx=0, dy=0, theta=None)`

Move drawing position relative to the reference point. Change drawing theta if provided.

Return type
None

`pop()`

Pop/load the drawing state. Location and angle are returned to previously pushed state.

Return type
None

push()

Push/save the drawing state. Drawing.here and Drawing.theta are saved.

Return type
None

save(*fname*, *transparent=True*, *dpi=72*)

Save figure to a file

Parameters

- **fname** (str) – Filename to save. In Matplotlib backend, the file type is automatically determined from extension (png, svg, jpg). SVG backend only supports saving SVG format.
- **transparent** (bool) – Save as transparent background, if available
- **dpi** (float) – Dots-per-inch for raster formats

Return type
None

set_anchor(*name*)

Define a Drawing anchor at the current drawing position

Return type
None

undo()

Removes previously added element

Return type
None

6.2 Element

class schemdraw.elements.**Element**(***kwargs*)

Standard circuit element.

Keyword Arguments are equivalent to calling setter methods.

anchors

Dictionary of anchor positions in element coordinates

absanchors

Dictionary of anchor positions in absolute drawing coordinates

segments

List of drawing primitives making up the element

transform

Transformation from element to drawing coordinates

absdrop

Drop position in drawing coordinates, set after the element is added to a drawing

defaults

Default parameters for the element

Anchor names are dynmically added as attributes after placing the element in a Drawing.

anchor(*anchor*)

Specify anchor for placement. The anchor will be aligned with the position specified by *at()* method.

Return type
Element

at(*xy*, *dx=0*, *dy=0*)

Set the element xy position

Parameters
xy (XY | tuple['Element', str]) – (x,y) position or tuple of (Element, anchorname)

Return type
Element

color(*color*)

Sets the element color

Parameters
color (str) – color name or hex value (ie '#FFFFFF')

Return type
Element

down()

Set the direction to down

Return type
Element

drop(*drop*)

Set the drop position - where to leave the current drawing position after placing this element

Return type
Element

fill(*color=True*)

Sets the element fill color.

Parameters

- **color** (bool | str) – Color string name or hex value, or
- **color.** (*True to fill with the element line*)

Return type
Element

flip()

Apply flip up/down

Return type
Element

get_bbox(*transform=False*, *includetext=True*)

Get element bounding box

Parameters

- **transform** – Apply the element transform to the bbox to get bounds in Drawing coordinates

- **includetext** – Consider text when calculating bounding box. Text width and height can vary by font, so this produces an estimate of bounds.

Returns
Corners of the bounding box, (xmin, ymin, xmax, ymax)

gradient_fill(*color1*, *color2*, *vertical=True*)

Fill the element with a linear gradient between two colors. Only supported in SVG backend.

Return type
Element

hold()

Do not move the Drawing *here* position after placing this element

Return type
Element

label(*label*, *loc=None*, *ofst=None*, *halign=None*, *valign=None*, *rotate=False*, *fontsize=None*, *font=None*, *mathfont=None*, *color=None*, *href=None*, *decoration=None*)

Add a label to the Element.

Parameters

- **label** (str | Sequence[str]) – The text string or list of strings. If list, each string will be evenly spaced along the element (e.g. ['-', 'V', '+'])
- **loc** (Optional[LabelLoc]) – Label position within the Element. Either ('top', 'bottom', 'left', 'right'), or the name of an anchor within the Element.
- **ofst** (XY | float | None) – Offset from default label position
- **halign** (Optional[Halign]) – Horizontal text alignment ('center', 'left', 'right')
- **valign** (Optional[Valign]) – Vertical text alignment ('center', 'top', 'bottom')
- **rotate** (bool | float) – True to rotate label with element, or specify rotation angle in degrees
- **fontsize** (Optional[float]) – Size of label font
- **font** (Optional[str]) – Name/font-family of label text
- **mathfont** (Optional[str]) – Name/font-family of math text
- **color** (Optional[str]) – Color of label
- **href** (Optional[str]) – Hyperline target (jump to)
- **decoration** (Optional[str]) – "underline" or "overline"

left()

Set the direction to left

Return type
Element

linestyle(*ls*)

Sets the element line style

Parameters
ls (Literal['-', ':', '--', '-.']) – Line style ('-', ':', '–', '-.').

Return type
Element

linewidth(*lw*)

Sets the element line width

Parameters
lw (float) – Line width

Return type
Element

reverse()

Apply reverse left/right

Return type
Element

right()

Set the direction to right

Return type
Element

scale(*scale=1*)

Apply scale/zoom factor to element

Return type
Element

scalex(*scale=1*)

Apply horizontal scale/zoom to element

Return type
Element

scaley(*scale=1*)

Apply vertical scale/zoom to element

Return type
Element

style(*color=None*, *fill=None*, *ls=None*, *lw=None*)

Apply all style parameters

Parameters

- **color** (Optional[str]) – Color string or hex value
- **fill** (Optional[str]) – Color string or hex
- **ls** (Optional[Literal['-', ':', '--', '-.']]) – Line style (‘-’, ‘:’, ‘–’, ‘-.’)
- **lw** (Optional[float]) – Line width

Return type
Element

theta(*theta*)

Set the drawing direction angle in degrees

Return type
Element

up()

Set the direction to up

Return type
Element

zorder(*zorder*)

Sets the element zorder. Higher zorders will be drawn above lower zorder elements.

Return type
Element

6.3 Element2Term

class schemdraw.elements.**Element2Term**(***kwargs*)

Two terminal element. The element leads can be automatically extended to the start and ending positions.

Anchors:

- start
- center
- end

dot(*open=False*)

Add a dot to the end of the element

Return type
Element2Term

down(*length=None*)

Set the direction to down

Return type
Element

endpoints(*start*, *end*)

Sets absolute endpoints of element

Return type
Element2Term

idot(*open=False*)

Add a dot to the input/start of the element

Return type
Element2Term

left(*length=None*)

Set the direction to left

Return type
Element

length(*length*)

Sets total length of element

Return type
Element2Term

right(*length=None*)

Set the direction to right

Return type
Element

shift(*shift*)

Shift the element within its leads to one end or the other. The shift parameter should be between -1 and 1, indicating fraction of the lead extension.

Return type
Element2Term

to(*xy*, *dx=0*, *dy=0*)

Sets ending position of element

Parameters

- **xy** (Union[Tuple[float, float], Point]) – Ending position of element
- **dx** (float) – X-offset from xy position
- **dy** (float) – Y-offset from xy position

Return type
Element2Term

tox(*x*)

Sets ending x-position of element (for horizontal elements)

Return type
Element2Term

toy(*y*)

Sets ending y-position of element (for vertical elements)

Return type
Element2Term

up(*length=None*)

Set the direction to up

Return type
Element

6.4 ElementDrawing

class schemdraw.elements.**ElementDrawing**(**args*, ***kwargs*)

Create an element from a Drawing

Parameters
drawing – The Drawing instance to convert to an element

6.5 ElementImage

class schemdraw.elements.**ElementImage**(*image*, *width*, *height*, *xy=(0, 0)*, *imgfmt=None*, ***kwargs*)

Element from an Image file

Parameters

- **image** (str | BinaryIO) – Image filename or open file pointer
- **width** (float) – Width to draw image in Drawing
- **height** (float) – Height to draw image in Drawing
- **xy** (Point) – Origin (lower left corner)

6.6 Element Style

schemdraw.elements.**style**(*style*)

Set global element style

Parameters

style – dictionary of {elementname: Element} to change the element module namespace. Use *elements.STYLE_US* or *elements.STYLE_IEC* to define U.S. or European/IEC element styles.

schemdraw.**config**(*unit=3.0*, *inches_per_unit=0.5*, *lblofst=0.1*, *fontsize=14.0*, *font='sans-serif'*, *color='black'*, *lw=2.0*, *ls='-'*, *fill=None*, *bgcolor=None*, *margin=0.1*, *mathfont=None*)

Set global schemdraw style configuration

Parameters

- **unit** (float) – Full length of a 2-terminal element. Inner zig-zag portion of a resistor is 1.0 units.
- **inches_per_unit** (float) – Inches per drawing unit for setting drawing scale
- **lblofst** (float) – Default offset between element and its label
- **fontsize** (float) – Default font size for text labels
- **font** (str) – Default font family for text labels
- **color** (str) – Default color name or RGB (0-1) tuple
- **lw** (float) – Default line width for elements
- **ls** (Literal['-', ':', '--', '-.']) – Default line style
- **fill** (Optional[str]) – Deault fill color for closed elements
- **margin** (float) – White space around the drawing in drawing units
- **mathont** – Font for math delimited by $..$

Return type

None

schemdraw.**theme**(*theme='default'*)

Set schemdraw theme (line color and background color). Themes match those in jupyter-themes package (https://github.com/dunovank/jupyter-themes).

Available themes:

- default (black on white)
- dark (white on black)
- solarizedd
- solarizedl
- onedork
- oceans16
- monokai
- gruvboxl
- gruvboxd
- grade3
- chesterish

schemdraw.**use**(*backend='matplotlib'*)

Change default backend, either 'matplotlib' or 'svg'

Return type
None

6.7 Segment Drawing Primitives

Schemdraw drawing segments.

Each element is made up of one or more segments that define drawing primitives.

class schemdraw.segments.**Segment**(*path*, *color=None*, *lw=None*, *ls=None*, *capstyle=None*, *joinstyle=None*, *fill=None*, *arrow=None*, *arrowwidth=0.15*, *arrowlength=0.25*, *clip=None*, *zorder=None*, *visible=True*)

A segment path

Parameters

- **path** (Sequence[XY]) – List of (x,y) coordinates making the path
- **color** (Optional[str | tuple[float, float, float]]) – Color for this segment
- **lw** (Optional[float]) – Line width for the segment
- **ls** (Optional[Linestyle]) – Line style for the segment '-', '–', ':', etc.
- **capstyle** (Optional[Capstyle]) – Capstyle for the segment: 'butt', 'round', 'square', ('projecting')
- **joinstyle** (Optional[Joinstyle]) – Joinstyle for the segment: 'round', 'miter', or 'bevel'
- **fill** (Optional[str]) – Color to fill if path is closed
- **arrow** (Optional[str]) – Arrowhead specifier, such as '->', '<-', '<->', '-o', etc.
- **arrowwidth** (float) – Width of arrowhead
- **arrowlength** (float) – Length of arrowhead
- **clip** (Optional[BBox]) – Bounding box to clip to
- **zorder** (Optional[int]) – Z-order for segment

- **visible** (bool) – Show the segment when drawn

doflip()

Vertically flip the element

Return type
None

doreverse(*centerx*)

Reverse the path (flip horizontal about the center of the path)

Return type
None

draw(*fig*, *transform*, ***style*)

Draw the segment

Parameters

- **fig** – schemdraw.Figure to draw on
- **transform** – Transform to apply before drawing
- **style** – Default style parameters

Return type
None

get_bbox()

Get bounding box (untransformed)

Returns
(xmin, ymin, xmax, ymax)

Return type
Bounding box limits

xform(*transform*, ***style*)

Return a new Segment that has been transformed to its global position

Parameters

- **transform** – Transformation to apply
- **style** – Style parameters from Element to apply as default

Return type
Segment

class schemdraw.segments.**SegmentArc**(*center*, *width*, *height*, *theta1=35*, *theta2=-35*, *arrow=None*, *arrowwidth=0.15*, *arrowlength=0.25*, *angle=0*, *color=None*, *lw=None*, *ls=None*, *fill=None*, *clip=None*, *zorder=None*, *visible=True*)

An elliptical arc drawing segment

Parameters

- **center** (XY) – Center of the arc ellipse
- **width** (float) – Width of the arc ellipse
- **height** (float) – Height of the arc ellipse
- **theta1** (float) – Starting angle in degrees

- **theta2** (float) – Ending angle in degrees
- **arrow** (Optional[Arcdirection]) – Direction of arrowhead ('cw' or 'ccw')
- **angle** (float) – Rotation of the ellipse defining the arc
- **color** (Optional[str | tuple[float, float, float]]) – Color for this segment
- **lw** (Optional[float]) – Line width for the segment
- **ls** (Optional[Linestyle]) – Line style for the segment
- **clip** (Optional[BBox]) – Bounding box to clip to
- **zorder** (Optional[int]) – Z-order for segment
- **visible** (bool) – Show the segment when drawn

doflip()

Vertically flip the element

Return type
None

doreverse(*centerx*)

Reverse the path (flip horizontal about the centerx point)

Return type
None

draw(*fig*, *transform*, ***style*)

Draw the segment

Parameters

- **fig** – schemdraw.Figure to draw on
- **transform** – Transform to apply before drawing
- **style** – Default style parameters

Return type
None

get_bbox()

Get bounding box (untransformed)

Return type
BBox

Returns
Bounding box limits (xmin, ymin, xmax, ymax)

xform(*transform*, ***style*)

Return a new Segment that has been transformed to its global position

Parameters

- **transform** – Transformation to apply
- **style** – Style parameters from Element to apply as default

Return type
SegmentArc

class schemdraw.segments.**SegmentBezier**(*p*, *color=None*, *lw=None*, *ls=None*, *capstyle=None*, *arrow=None*, *arrowlength=0.25*, *arrowwidth=0.15*, *clip=None*, *zorder=None*, *visible=True*)

Quadratic or Cubic Bezier curve segment

Parameters

- **p** (Sequence[XY]) – control points (3 or 4)
- **color** (Optional[str | tuple[float, float, float]]) – Color for this segment
- **lw** (Optional[float]) – Line width for the segment
- **ls** (Optional[Linestyle]) – Line style for the segment '-', '–', ':', etc.
- **capstyle** (Optional[Capstyle]) – Capstyle for the segment: 'butt', 'round', 'square', ('projecting')
- **joinstyle** – Joinstyle for the segment: 'round', 'miter', or 'bevel'
- **fill** – Color to fill if path is closed
- **arrow** (Optional[str]) – Arrowhead specifier, such as '->', '<-', or '<->'
- **arrowwidth** (float) – Width of arrowhead
- **arrowlength** (float) – Length of arrowhead
- **clip** (Optional[BBox]) – Bounding box to clip to
- **zorder** (Optional[int]) – Z-order for segment
- **visible** (bool) – Show the segment when drawn

doflip()

Vertically flip the element

Return type
None

doreverse(*centerx*)

Reverse the path (flip horizontal about the centerx point)

Return type
None

draw(*fig*, *transform*, ***style*)

Draw the segment

Parameters

- **fig** – schemdraw.Figure to draw on
- **transform** – Transform to apply before drawing
- **style** – Default style parameters

Return type
None

get_bbox()

Get bounding box (untransformed)

Return type
BBox

Returns
Bounding box limits (xmin, ymin, xmax, ymax)

xform(*transform*, ***style*)

Return a new Segment that has been transformed to its global position

Parameters

- **transform** – Transformation to apply
- **style** – Style parameters from Element to apply as default

Return type
SegmentBezier

class schemdraw.segments.**SegmentCircle**(*center*, *radius*, *color=None*, *lw=None*, *ls=None*, *fill=None*, *clip=None*, *zorder=None*, *ref=None*, *visible=True*)

A circle drawing segment

Parameters

- **center** (XY) – (x, y) center of the circle
- **radius** (float) – Radius of the circle
- **color** (Optional[str | tuple[float, float, float]]) – Color for this segment
- **lw** (Optional[float]) – Line width for the segment
- **ls** (Optional[Linestyle]) – Line style for the segment
- **fill** (bool | str | None) – Color to fill if path is closed. True -> fill with element color.
- **clip** (Optional[BBox]) – Bounding box to clip to
- **zorder** (Optional[int]) – Z-order for segment
- **ref** (Optional[EndRef]) – Flip reference ['start', 'end', None].
- **visible** (bool) – Show the segment when drawn

doflip()

Flip the segment up/down

Return type
None

doreverse(*centerx*)

Reverse the path (flip horizontal about the centerx point)

Return type
None

draw(*fig*, *transform*, ***style*)

Draw the segment

Parameters

- **fig** – schemdraw.Figure to draw on
- **transform** – Transform to apply before drawing
- **style** – Default style parameters

Return type
None

get_bbox()

Get bounding box (untransformed)

Return type
BBox

Returns
Bounding box limits (xmin, ymin, xmax, ymax)

xform(*transform*, ***style*)

Return a new Segment that has been transformed to its global position

Parameters

- **transform** – Transformation to apply
- **style** – Style parameters from Element to apply as default

Return type
SegmentCircle' | 'SegmentArc

class schemdraw.segments.**SegmentImage**(*image*, *xy=(0, 0)*, *width=3*, *height=1*, *rotate=0*, *imgfmt=None*, *zorder=None*)

PNG or SVG Image

doflip()

Vertically flip the element

Return type
None

doreverse(*centerx*)

Reverse the image

Return type
None

draw(*fig*, *transform*, ***style*)

Draw the image

Parameters

- **fig** – schemdraw.Figure to draw on
- **transform** – Transform to apply before drawing
- **style** – Default style parameters

Return type
None

get_bbox()

Get bounding box (untransformed)

Return type
BBox

Returns
Bounding box limits (xmin, ymin, xmax, ymax)

xform(*transform*, ***style*)

Return a new Segment that has been transformed to its global position

Parameters

- **transform** – Transformation to apply
- **style** – Style parameters from Element to apply as default

Return type
SegmentImage

class schemdraw.segments.**SegmentPath**(*path*, *color=None*, *lw=None*, *ls=None*, *capstyle=None*, *joinstyle=None*, *fill=None*, *clip=None*, *zorder=None*, *visible=True*)

Segment defined like svg <path> element made of M, L, C, Q, Z, etc....

doflip()

Vertically flip the element

Return type
None

doreverse(*centerx*)

Reverse the path (flip horizontal about the center of the path)

Return type
None

draw(*fig*, *transform*, ***style*)

Draw the segment

Parameters

- **fig** – schemdraw.Figure to draw on
- **transform** – Transform to apply before drawing
- **style** – Default style parameters

Return type
None

get_bbox()

Get bounding box (untransformed)

Returns
(xmin, ymin, xmax, ymax)

Return type
Bounding box limits

xform(*transform*, ***style*)

Return a new SegmentPath that has been transformed to its global position

Parameters

- **transform** – Transformation to apply
- **style** – Style parameters from Element to apply as default

Return type
SegmentPath

class schemdraw.segments.**SegmentPoly**(*verts*, *closed=True*, *cornerradius=0*, *color=None*, *fill=None*, *lw=None*, *ls=None*, *hatch=False*, *joinstyle=None*, *capstyle=None*, *clip=None*, *zorder=None*, *visible=True*)

A polygon segment

Parameters

- **xy** – List of (x,y) coordinates making the polygon
- **closed** (bool) – Draw a closed polygon (default True)
- **cornerradius** (float) – Round the corners to this radius (0 for no rounding)
- **color** (Optional[str | tuple[float, float, float]]) – Color for this segment
- **fill** (Optional[str]) – Color to fill if path is closed
- **lw** (Optional[float]) – Line width for the segment
- **ls** (Optional[Linestyle]) – Line style for the segment
- **hatch** (bool) – Show hatch lines
- **capstyle** (Optional[Capstyle]) – Capstyle for the segment: 'butt', 'round', 'square', ('projecting')
- **joinstyle** (Optional[Joinstyle]) – Joinstyle for the segment: 'round', 'miter', or 'bevel'
- **clip** (Optional[BBox]) – Bounding box to clip to
- **zorder** (Optional[int]) – Z-order for segment
- **visible** (bool) – Show the segment when drawn

doflip()

Vertically flip the element

Return type
None

doreverse(*centerx*)

Reverse the path (flip horizontal about the centerx point)

Return type
None

draw(*fig*, *transform*, ***style*)

Draw the segment

Parameters

- **fig** – schemdraw.Figure to draw on
- **transform** – Transform to apply before drawing
- **style** – Default style parameters

Return type
None

get_bbox()

Get bounding box (untransformed)

Return type
BBox

Returns
Bounding box limits (xmin, ymin, xmax, ymax)

xform(*transform*, ***style*)

Return a new Segment that has been transformed to its global position

Parameters

- **transform** – Transformation to apply
- **style** – Style parameters from Element to apply as default

Return type
SegmentPoly

class schemdraw.segments.**SegmentText**(*pos*, *label*, *align=None*, *rotation=None*, *rotation_mode=None*, *rotation_global=True*, *color=None*, *bgcolor=None*, *fontsize=14*, *font=None*, *mathfont=None*, *clip=None*, *zorder=None*, *visible=True*, *href=None*, *decoration=None*)

A text drawing segment

Parameters

- **pos** (XY) – (x, y) coordinates for text
- **label** (str) – Text to draw
- **align** (Optional[tuple[Halign, Valign]]) – Tuple of (horizontal, vertical) alignment where horizontal is ('center', 'left', 'right') and vertical is ('center', 'top', 'bottom')
- **rotation** (Optional[float]) – Rotation angle in degrees
- **rotation_mode** (Optional[RotationMode]) – See Matplotlib documentation. 'anchor' or 'default'.
- **rotation_global** (bool) – Lock rotation to world rather than component. Defaults to True
- **color** (Optional[str | tuple[float, float, float]]) – Color for this segment
- **bgcolor** (Optional[str | tuple[float, float, float]]) – Background color under the text
- **fontsize** (float) – Font size
- **font** (Optional[str]) – Font name/family
- **mathfont** (Optional[str]) – Math font name/family
- **clip** (Optional[BBox]) – Bounding box to clip to
- **zorder** (Optional[int]) – Z-order for segment
- **visible** (bool) – Show the segment when drawn

doflip()

Vertically flip the element

Return type
None

doreverse(*centerx*)

Reverse the path (flip horizontal about the centerx point)

Return type
None

draw(*fig*, *transform*, ***style*)

Draw the segment

Parameters

- **fig** – schemdraw.Figure to draw on
- **transform** – Transform to apply before drawing
- **style** – Default style parameters

Return type
None

get_bbox()

Get bounding box (untransformed)

Return type
BBox

Returns
Bounding box limits (xmin, ymin, xmax, ymax)

xform(*transform*, ***style*)

Return a new Segment that has been transformed to its global position

Parameters

- **transform** – Transformation to apply
- **style** – Style parameters from Element to apply as default

Return type
SegmentText

6.8 Electrical Elements

6.8.1 Two-Terminal Elements

Two-terminal element definitions

class schemdraw.elements.twoterm.**Breaker**(*, *dots=None*, ***kwargs*)

Circuit breaker

Keyword Arguments

- **dots** – Show connection dots
- **arc_lw** – Line width of breaker arc
- **arc_color** – Color of breaker arc

class schemdraw.elements.twoterm.**CPE**(**args*, ***kwargs*)

Constant Phase Element

class schemdraw.elements.twoterm.**Capacitor**(*, *polar=None*, ***kwargs*)

Parameters
polar (Optional[bool]) – Add polarity + sign

class schemdraw.elements.twoterm.**Capacitor2**(*, *polar=None*, ***kwargs*)
Capacitor (curved bottom plate)

Parameters
polar (Optional[bool]) – Add polarity + sign

class schemdraw.elements.twoterm.**CapacitorTrim**(**args*, ***kwargs*)
Trim capacitor

Keyword Arguments

- **trim_lw** – Line width of trim
- **trim_color** – Color of trim

class schemdraw.elements.twoterm.**CapacitorVar**(**args*, ***kwargs*)
Variable capacitor

Keyword Arguments

- **arrowwidth** – Width of arrowhead
- **arrowlength** – Length of arrowhead
- **arrow_lw** – Line width of arrow
- **arrow_color** – Color of arrow

class schemdraw.elements.twoterm.**Crystal**(**args*, ***kwargs*)
Crystal oscillator

class schemdraw.elements.twoterm.**CurrentMirror**(**args*, ***kwargs*)
Current mirror with optional common terminal

Anchors:

- scommon

class schemdraw.elements.twoterm.**Diac**(**args*, ***kwargs*)
Diac (diode for alternating current)

class schemdraw.elements.twoterm.**Diode**(**args*, ***kwargs*)

class schemdraw.elements.twoterm.**DiodeShockley**(**args*, ***kwargs*)
Shockley Diode

class schemdraw.elements.twoterm.**DiodeTVS**(**args*, ***kwargs*)
Transient-Voltage Suppression (TVS) Diode

class schemdraw.elements.twoterm.**DiodeTunnel**(**args*, ***kwargs*)
Tunnel Diode

schemdraw.elements.twoterm.**Fuse**
alias of *FuseIEEE*

class schemdraw.elements.twoterm.**FuseIEC**(**args*, ***kwargs*)
Fuse (IEC Style)

class schemdraw.elements.twoterm.**FuseIEEE**(**args*, ***kwargs*)
Fuse (IEEE Style)

class schemdraw.elements.twoterm.**FuseUS**(*, *dots=None*, ***kwargs*)

Fuse (U.S. Style)

Parameters
dots (Optional[bool]) – Show dots on connections to fuse

fill(*color=True*)

Set element fill

Return type
Element

class schemdraw.elements.twoterm.**Inductor**(*core=0*, ***kwargs*)

Keyword Arguments

- **core** – Number of core lines to draw parallel to inductor
- **core_ls** – Line style of core
- **core_lw** – Line width of core
- **core_color** – Color of core

class schemdraw.elements.twoterm.**Inductor2**(*, *loops=None*, *core=0*, ***kwargs*)

Inductor, drawn as cycloid (loopy)

Keyword Arguments

- **loops** – Number of inductor loops [default: 4]
- **core** – Number of core lines to draw parallel to inductor
- **core_ls** – Line style of core
- **core_lw** – Line width of core
- **core_color** – Color of core

class schemdraw.elements.twoterm.**Josephson**(*, *box=None*, ***kwargs*)

Josephson Junction

Parameters
box (Optional[bool]) – Draw a square around the Josephson X symbol

class schemdraw.elements.twoterm.**LED**(**args*, ***kwargs*)

Light emitting diod

Keyword Arguments

- **arrowwidth** – Width of arrowhead
- **arrowlength** – Length of arrowhead
- **arrow_lw** – Line width of arrow
- **arrow_color** – Color of arrow

class schemdraw.elements.twoterm.**LED2**(**args*, ***kwargs*)

Light emitting diode (curvy light lines)

Keyword Arguments

- **arrowwidth** – Width of arrowhead
- **arrowlength** – Length of arrowhead

- **arrow_lw** – Line width of arrow
- **arrow_color** – Color of arrow

class schemdraw.elements.twoterm.**Memristor**(**args*, ***kwargs*)
Memristor (resistor with memory)

class schemdraw.elements.twoterm.**Memristor2**(**args*, ***kwargs*)
Memristor (resistor with memory), alternate style

class schemdraw.elements.twoterm.**Norator**(**args*, ***kwargs*)

class schemdraw.elements.twoterm.**Nullator**(**args*, ***kwargs*)
This element does not support filling

class schemdraw.elements.twoterm.**Photodiode**(**args*, ***kwargs*)
Photo-sensitive diode

Keyword Arguments

- **arrowwidth** – Width of arrowhead
- **arrowlength** – Length of arrowhead
- **arrow_lw** – Line width of arrow
- **arrow_color** – Color of arrow

schemdraw.elements.twoterm.**Photoresistor**
alias of *PhotoresistorIEEE*

schemdraw.elements.twoterm.**PhotoresistorBox**
alias of *PhotoresistorIEC*

class schemdraw.elements.twoterm.**PhotoresistorIEC**(**args*, ***kwargs*)
Photo-resistor (European style)

Keyword Arguments

- **arrowwidth** – Width of arrowhead
- **arrowlength** – Length of arrowhead
- **arrow_lw** – Line width of arrow
- **arrow_color** – Color of arrow

class schemdraw.elements.twoterm.**PhotoresistorIEEE**(**args*, ***kwargs*)
Photo-resistor (U.S. style)

Keyword Arguments

- **arrowwidth** – Width of arrowhead
- **arrowlength** – Length of arrowhead
- **arrow_lw** – Line width of arrow
- **arrow_color** – Color of arrow

schemdraw.elements.twoterm.**PotBox**
alias of *PotentiometerIEC*

schemdraw.elements.twoterm.**Potentiometer**
alias of *PotentiometerIEEE*

class schemdraw.elements.twoterm.**PotentiometerIEC**(**args*, ***kwargs*)

Potentiometer (European style)

Anchors:

tap

Keyword Arguments

- **arrowwidth** – Width of arrowhead
- **arrowlength** – Length of arrowhead
- **tap_length** – Length of tap line

class schemdraw.elements.twoterm.**PotentiometerIEEE**(**args*, ***kwargs*)

Potentiometer (U.S. style)

Anchors:

tap

Keyword Arguments

- **arrowwidth** – Width of arrowhead
- **arrowlength** – Length of arrowhead
- **tap_length** – Length of tap line

schemdraw.elements.twoterm.**RBox**

alias of *ResistorIEC*

schemdraw.elements.twoterm.**RBoxVar**

alias of *ResistorVarIEC*

schemdraw.elements.twoterm.**Resistor**

alias of *ResistorIEEE*

class schemdraw.elements.twoterm.**ResistorIEC**(**args*, ***kwargs*)

Resistor as box (IEC/European style)

class schemdraw.elements.twoterm.**ResistorIEEE**(**args*, ***kwargs*)

Resistor (IEEE/U.S. style)

schemdraw.elements.twoterm.**ResistorVar**

alias of *ResistorVarIEEE*

class schemdraw.elements.twoterm.**ResistorVarIEC**(**args*, ***kwargs*)

Variable resistor (European style)

Keyword Arguments

- **arrowwidth** – Width of arrowhead
- **arrowlength** – Length of arrowhead
- **arrow_lw** – Line width of arrow
- **arrow_color** – Color of arrow

class schemdraw.elements.twoterm.**ResistorVarIEEE**(**args*, ***kwargs*)
Variable resistor (U.S. style)

Keyword Arguments

- **arrowwidth** – Width of arrowhead
- **arrowlength** – Length of arrowhead
- **arrow_lw** – Line width of arrow
- **arrow_color** – Color of arrow

class schemdraw.elements.twoterm.**Rshunt**(**args*, ***kwargs*)
Shunt Resistor

class schemdraw.elements.twoterm.**SCR**(**args*, ***kwargs*)
Silicon controlled rectifier (or thyristor)

Anchors:
gate

class schemdraw.elements.twoterm.**Schottky**(**args*, ***kwargs*)
Schottky Diode

class schemdraw.elements.twoterm.**SparkGap**(**args*, ***kwargs*)
Spark Gap

Keyword Arguments

- **gap** – Width of gap between arrows
- **arrowwidth** – Width of arrowheads
- **arrowlength** – Length of arrowheads

class schemdraw.elements.twoterm.**Thermistor**(**args*, ***kwargs*)

class schemdraw.elements.twoterm.**Triac**(**args*, ***kwargs*)

Anchors:
gate

class schemdraw.elements.twoterm.**Varactor**(**args*, ***kwargs*)
Varactor Diode/Varicap/Variable Capacitance Diode

class schemdraw.elements.twoterm.**VoltageMirror**(**args*, ***kwargs*)
Voltage mirror with optional common terminal

This element does not support filling

Anchors:

- scommon

class schemdraw.elements.twoterm.**Zener**(**args*, ***kwargs*)
Zener Diode

Sources, meters, and lamp elements

class schemdraw.elements.sources.**Battery**(**args*, ***kwargs*)

class schemdraw.elements.sources.**BatteryCell**(**args*, ***kwargs*)
Cell

class schemdraw.elements.sources.**BatteryDouble**(**args*, ***kwargs*)

Double-stack Battery

class schemdraw.elements.sources.**Lamp**(**args*, ***kwargs*)

Incandescent Lamp

Keyword Arguments

- **filament_lw** – Line width of filament
- **filament_color** – Color of filament

class schemdraw.elements.sources.**Lamp2**(**args*, ***kwargs*)

Incandescent Lamp (with X through a Source)

Keyword Arguments

- **filament_lw** – Line width of filament
- **filament_color** – Color of filament

class schemdraw.elements.sources.**MeterA**(**args*, ***kwargs*)

Ammeter

class schemdraw.elements.sources.**MeterAnalog**(*inputs=True*, *needle_percent=70*, ***kwargs*)

Analog meter drawn as a box

Parameters

- **inputs** (bool) – Show input connections
- **needle_percent** (float) – Position of the needle along the window from 0 to 100
- **needle_color** – Color of the needle
- **needle_width** – Line width of the needle
- **window_fill** – Fill color for window area
- **input_ofst** – (x, y) offset of input connectors
- **input_lw** – Line width of input connector circles
- **input_rad** – Radius of input connector circles
- **input_fill** – Fill color of input connector circles

class schemdraw.elements.sources.**MeterArrow**(**args*, ***kwargs*)

Meter with diagonal arrow

Keyword Arguments

- **arrowwidth** – Width of arrowhead
- **arrowlength** – length of arrowhead
- **arrow_lw** – Line width of arrow
- **arrow_color** – Color of arrow

class schemdraw.elements.sources.**MeterBox**(*inputs=True*, ***kwargs*)

Base class for square meter/scope boxes

Keyword Arguments

- **width** – Width and height of square box

- **corner** – Corner radius
- **input_ofst** – Distance from corner to input connections
- **input_lw** – line width of connector circles
- **input_rad** – radius of connector circles
- **input_fill** – Fill coloro of connector circles

class schemdraw.elements.sources.**MeterDigital**(*inputs=True*, ***kwargs*)

Digital meter drawn as a box

Parameters

- **inputs** (bool) – Show input connections
- **screen_fill** – Fill color for screen area
- **screen_lw** – Line width of the screen
- **input_ofst** – (x, y) offset of input connectors
- **input_lw** – Line width of input connector circles
- **input_rad** – Radius of input connector circles
- **input_fill** – Fill color of input connector circles

class schemdraw.elements.sources.**MeterI**(**args*, ***kwargs*)

Current Meter (I)

class schemdraw.elements.sources.**MeterOhm**(**args*, ***kwargs*)

Ohm meter

class schemdraw.elements.sources.**MeterV**(**args*, ***kwargs*)

Volt meter

class schemdraw.elements.sources.**Neon**(**args*, ***kwargs*)

Neon bulb

class schemdraw.elements.sources.**Oscilloscope**(*signal='none'*, *inputs=True*, ***kwargs*)

Oscilloscope drawn as a box

Parameters

- **signal** (str) – Type of signal to display on the screen. 'sine', 'square', or 'triangle'
- **inputs** (bool) – Show input connections
- **screen_fill** – Fill color for screen area
- **screen_lw** – Line width of the screen
- **grid** – Whether to show grid over the screen
- **grid_color** – Color for the oscope grid
- **grid_lw** – Line width of the grid
- **signal_lw** – Line width of the displayed signal
- **signal_color** – Color of the displayed signal
- **input_ofst** – (x, y) offset of input connectors
- **input_lw** – Line width of input connector circles

- **input_rad** – Radius of input connector circles
- **input_fill** – Fill color of input connector circles

class schemdraw.elements.sources.**Solar**(**args*, ***kwargs*)

Solar source

Keyword Arguments

- **arrowwidth** – Width of arrowhead
- **arrowlength** – length of arrowhead
- **arrow_lw** – Line width of arrow
- **arrow_color** – Color of arrow

class schemdraw.elements.sources.**Source**(**args*, ***kwargs*)

Generic source element

class schemdraw.elements.sources.**SourceControlled**(**args*, ***kwargs*)

Generic controlled source

class schemdraw.elements.sources.**SourceControlledI**(**args*, ***kwargs*)

Controlled current source

Keyword Arguments

- **arrowwidth** – Width of arrowhead
- **arrowlength** – length of arrowhead
- **arrow_lw** – Line width of arrow
- **arrow_color** – Color of arrow

class schemdraw.elements.sources.**SourceControlledV**(**args*, ***kwargs*)

Controlled voltage source

class schemdraw.elements.sources.**SourceI**(**args*, ***kwargs*)

Current source

Keyword Arguments

- **arrowwidth** – Width of arrowhead
- **arrowlength** – length of arrowhead
- **arrow_lw** – Line width of arrow
- **arrow_color** – Color of arrow

class schemdraw.elements.sources.**SourcePulse**(**args*, ***kwargs*)

Pulse source

Keyword Arguments

- **pulse_lw** – Line width of pulse curve
- **pulse_color** – Color of pulse curve

class schemdraw.elements.sources.**SourceRamp**(**args*, ***kwargs*)

Ramp/sawtooth source

Keyword Arguments

- **ramp_lw** – Line width of ramp curve

- **ramp_color** – Color of ramp curve

class schemdraw.elements.sources.**SourceSin**(**args*, ***kwargs*)

Source with sine

Keyword Arguments

- **sin_lw** – Line width of sine curve
- **sin_color** – Color of sine curve

class schemdraw.elements.sources.**SourceSquare**(**args*, ***kwargs*)

Square wave source

Keyword Arguments

- **square_lw** – Line width of square curve
- **square_color** – Color of square curve

class schemdraw.elements.sources.**SourceTriangle**(**args*, ***kwargs*)

Triangle source

Keyword Arguments

- **tri_lw** – Line width of triangle curve
- **tri_color** – Color of triangle curve

class schemdraw.elements.sources.**SourceV**(**args*, ***kwargs*)

Voltage source

6.8.2 One-terminal Elements

One terminal element definitions

class schemdraw.elements.oneterm.**Antenna**(**args*, ***kwargs*)

class schemdraw.elements.oneterm.**AntennaLoop**(**args*, ***kwargs*)

Loop antenna (diamond style)

class schemdraw.elements.oneterm.**AntennaLoop2**(**args*, ***kwargs*)

Loop antenna (square style)

class schemdraw.elements.oneterm.**Ground**(*, *lead=None*, ***kwargs*)

Ground connection

Keyword Arguments
lead – Show lead wire [default: True]

class schemdraw.elements.oneterm.**GroundChassis**(**args*, ***kwargs*)

Chassis ground

Keyword Arguments
lead – Show lead wire

class schemdraw.elements.oneterm.**GroundSignal**(*, *lead=True*, ***kwargs*)

Signal ground

Keyword Arguments
lead – Show lead wire

class schemdraw.elements.oneterm.**NoConnect**(**args*, ***kwargs*)

No Connection

class schemdraw.elements.oneterm.**Vdd**(**args*, ***kwargs*)

Vdd connection

Keyword Arguments
lead – Show lead wire

class schemdraw.elements.oneterm.**Vss**(**args*, ***kwargs*)

Vss connection

Keyword Arguments
lead – Show lead wire

6.8.3 Switches

Switches and buttons

class schemdraw.elements.switches.**Button**(*nc=False*, *contacts=True*, ***kwargs*)

Push button switch

Parameters

- **nc** (bool) – Normally closed
- **contacts** (bool) – Draw contacts as open circles

class schemdraw.elements.switches.**Switch**(*action=None*, *contacts=True*, *nc=False*, ***kwargs*)

Toggle Switch

Parameters

- **action** (Optional[Literal['open', 'close']]) – action arrow ('open' or 'close')
- **contacts** (bool) – Draw contacts as open circles
- **nc** (bool) – Draw normally closed contact line
- **arrowwidth** – Width of arrowhead
- **arrowlength** – Length of arrowhead
- **arrow_lw** – line width of arrow
- **arrow_color** – Color of arrow

class schemdraw.elements.switches.**SwitchDIP**(***, *n=3*, *pattern=None*, *switchcolor=None*, *swidth=None*, *spacing=None*, ***kwargs*)

DIP switch

Parameters

- **n** (int) – Number of switches
- **pattern** (Optional[Sequence[bool]]) – Boolean sequence indicating whether each switch is flipped up or down
- **switchcolor** (Optional[str]) – Fill color for flipped switches [default: #333333]
- **swidth** (Optional[float]) – Width of one switch [default: 0.4]
- **spacing** (Optional[float]) – Spacing between switches [default: 0.2]

class schemdraw.elements.switches.**SwitchDpdt**(*link=True*, *contacts=True*, ***kwargs*)

Double-pole double-throw switch

Parameters

- **link** (bool) – Show dotted line linking switch levers
- **contacts** (bool) – Draw contacts as open circles

Anchors:

- p1
- p2
- t1
- t2
- t3
- t4

class schemdraw.elements.switches.**SwitchDpst**(*link=True*, *contacts=True*, ***kwargs*)

Double-pole single-throw switch

Parameters

- **link** (bool) – Show dotted line linking switch levers
- **contacts** (bool) – Draw contacts as open circles

Anchors:

- p1
- p2
- t1
- t2

class schemdraw.elements.switches.**SwitchReed**(**args*, ***kwargs*)

Reed Switch

class schemdraw.elements.switches.**SwitchRotary**(***, *n=4*, *dtheta=None*, *theta0=None*, *radius=1*, *arrowlen=0.75*, *arrowcontact=0*, ***kwargs*)

Rotary Switch

Parameters

- **n** (int) – number of contacts
- **dtheta** (Optional[float]) – angle in degrees between each contact
- **theta0** (Optional[float]) – angle in degrees of first contact
- **radius** (float) – radius of switch
- **arrowlen** (float) – length of switch arrow
- **arrowcontact** (int) – index of contact to point to

Values for dtheta and theta will be calculated based on *n* if not provided.

Anchors:

- P
- T[x] for each contact (starting at 1)

class schemdraw.elements.switches.**SwitchSpdt**(*action=None*, *contacts=True*, ***kwargs*)

Single-pole double throw switch.

Parameters

- **action** (Optional[Literal['open', 'close']]) – action arrow ('open' or 'close')
- **contacts** (bool) – Draw contacts as open circles

Anchors:

- a
- b
- c

class schemdraw.elements.switches.**SwitchSpdt2**(*action=None*, *contacts=True*, ***kwargs*)

Single-pole double throw switch, throws above and below.

Parameters

- **action** (Optional[Literal['open', 'close']]) – action arrow ('open' or 'close')
- **contacts** (bool) – Draw contacts as open circles
- **arrowwidth** – Width of arrowhead
- **arrowlength** – Length of arrowhead
- **arrow_lw** – line width of arrow
- **arrow_color** – Color of arrow

Anchors:

- a
- b
- c

6.8.4 Lines

Lines, Arrows, and Labels

class schemdraw.elements.lines.**Annotate**(*k=0.75*, *th1=75*, *th2=180*, *arrow='<-'*, ***, *arrowlength=None*, *arrowwidth=None*, ***kwargs*)

Draw a curved arrow pointing to *at* position, ending at *to* position, with label location at the tail of the arrow (See also *Arc3*).

Parameters

- **k** (float) – Control point factor. Higher k means tighter curve.
- **th1** (float) – Angle at which the arc leaves start point
- **th2** (float) – Angle at which the arc leaves end point
- **arrow** (str) – arrowhead specifier, such as '->', '<-', '<->', or '-o'

- **arrowlength** (Optional[float]) – Length of arrowhead [default: 0.25]
- **arrowwidth** (Optional[float]) – Width of arrowhead [default: 0.2]

class schemdraw.elements.lines.**Arc2**(*k=0.5*, *arrow=None*, ***kwargs*)

Arc Element

Use *at* and *to* methods to define endpoints.

Arc2 is a quadratic Bezier curve with control point halfway between the endpoints, generating a symmetric 'C' curve.

Parameters

- **k** (float) – Control point factor. Higher k means more curvature.
- **arrow** (Optional[str]) – arrowhead specifier, such as '->', '<-', '<->', or '-o'

Anchors:
start end ctrl mid

delta(*dx=0*, *dy=0*)

Specify change in position

Return type
Element

to(*xy*, *dx=0*, *dy=0*)

Specify ending position

Parameters

- **xy** (Union[Tuple[float, float], Point]) – Ending position of element
- **dx** (float) – X-offset from xy position
- **dy** (float) – Y-offset from xy position

Return type
Element

class schemdraw.elements.lines.**Arc3**(*k=0.75*, *th1=0.0*, *th2=180*, *arrow=None*, *, *arrowlength=None*, *arrowwidth=None*, ***kwargs*)

Arc Element

Use *at* and *to* methods to define endpoints.

Arc3 is a cubic Bezier curve. Control points are set to extend the curve at the given angle for each endpoint.

Parameters

- **k** (float) – Control point factor. Higher k means tighter curve.
- **th1** (float) – Angle at which the arc leaves start point
- **th2** (float) – Angle at which the arc leaves end point
- **arrow** (Optional[str]) – arrowhead specifier, such as '->', '<-', '<->', or '-o'
- **arrowlength** (Optional[float]) – Length of arrowhead [default: 0.25]
- **arrowwidth** (Optional[float]) – Width of arrowhead [default: 0.2]

Anchors:
start end center ctrl1 ctrl2

delta(*dx=0*, *dy=0*)

Specify change in position

Return type
Element

to(*xy*, *dx=0*, *dy=0*)

Specify ending position

Parameters

- **xy** (Union[Tuple[float, float], Point]) – Ending position of element
- **dx** (float) – X-offset from xy position
- **dy** (float) – Y-offset from xy position

Return type
Element

class schemdraw.elements.lines.**ArcLoop**(*radius=0.6*, *arrow=None*, ***, *arrowlength=None*, *arrowwidth=None*, ***kwargs*)

Loop Arc

Use *at* and *to* methods to define endpoints.

ArcLoop is an arc drawn as part of a circle.

Parameters

- **radius** (float) – Radius of the arc
- **arrow** (Optional[str]) – arrowhead specifier, such as '->', '<-', '<->', or '-o' [default: None]
- **arrowlength** (Optional[float]) – Length of arrowhead [default: 0.25]
- **arrowwidth** (Optional[float]) – Width of arrowhead [default: 0.2]

Anchors:
start end mid BL BR TL TR

delta(*dx=0*, *dy=0*)

Specify ending position relative to start position

Return type
Element

to(*xy*, *dx=0*, *dy=0*)

Specify ending position

Parameters

- **xy** (Union[Tuple[float, float], Point]) – Ending position of element
- **dx** (float) – X-offset from xy position
- **dy** (float) – Y-offset from xy position

Return type
Element

class schemdraw.elements.lines.**ArcN**(*k=0.75*, *arrow=None*, *, *arrowlength=None*, *arrowwidth=None*, ***kwargs*)

N-Curve Arc

Use *at* and *to* methods to define endpoints.

ArcN approaches the endpoints vertically, leading to a 'N' shaped curve

Parameters

- **k** (float) – Control point factor. Higher k means tighter curve.
- **arrow** (Optional[str]) – arrowhead specifier, such as '->', '<-', '<->', or '-o' [default: None]
- **arrowlength** (Optional[float]) – Length of arrowhead [default: 0.25]
- **arrowwidth** (Optional[float]) – Width of arrowhead [default: 0.2]

class schemdraw.elements.lines.**ArcZ**(*k=0.75*, *arrow=None*, *, *arrowlength=None*, *arrowwidth=None*, ***kwargs*)

Z-Curve Arc

Use *at* and *to* methods to define endpoints.

ArcZ approaches the endpoints horizontally, leading to a 'Z' shaped curve

Parameters

- **k** (float) – Control point factor. Higher k means tighter curve.
- **arrow** (Optional[str]) – arrowhead specifier, such as '->', '<-', '<->', or '-o' [default: None]
- **arrowlength** (Optional[float]) – Length of arrowhead [default: 0.25]
- **arrowwidth** (Optional[float]) – Width of arrowhead [default: 0.2]

class schemdraw.elements.lines.**Arrow**(*, *double=False*, *arrowwidth=None*, *arrowlength=None*, ***kwargs*)

Parameters
double (bool) – Show arrowhead on both ends

Keyword Arguments

- **arrow** – arrowhead specifier, such as '->', '<-', '<->', '-o', or '|->'
- **arrowwidth** – Width of arrow head [default: 0.15]
- **arrowlength** – Length of arrow head [default: 0.25]

class schemdraw.elements.lines.**Arrowhead**(**args*, ***kwargs*)

Parameters

- **headwidth** – width of arrow head [default: .15]
- **headlength** – length of arrow head [default: .25]

class schemdraw.elements.lines.**CurrentLabel**(*, *length=None*, *top=None*, *reverse=None*, *ofst=None*, *headlength=None*, *headwidth=None*, ***kwargs*)

Current label arrow drawn above an element

Use *.at()* method to place the label over an existing element.

Parameters

- **reverse** (Optional[bool]) – Reverse the arrow direction
- **ofst** (Optional[float]) – Offset distance from centerline of element [default: 0.4]
- **length** (Optional[float]) – Length of the arrow [default: 2]
- **top** (Optional[bool]) – Draw arrow on top or bottom of element [default: True]
- **headlength** (Optional[float]) – Length of arrowhead [default: 0.3]
- **headwidth** (Optional[float]) – Width of arrowhead [default: 0.2]

at(*xy*, *dx=0*, *dy=0*)

Specify CurrentLabel position.

If xy is an Element, arrow will be centered along element and its color will also be inherited.

Parameters

- **xy** (XY | tuple['Element', str]) – The absolute (x, y) position or an
- **over** (*Element instance to center the arrow*)

Return type
Element

class schemdraw.elements.lines.**CurrentLabelInline**(*direction='in'*, *ofst=0.8*, *start=True*, ***, *headlength=None*, *headwidth=None*, ***kwargs*)

Current direction arrow, inline with element.

Use *.at()* method to place arrow on an Element instance

Parameters

- **direction** (Literal['in', 'out']) – arrow direction 'in' or 'out' of element
- **ofst** (float) – Offset along lead length
- **start** (bool) – Arrow at start or end of element
- **headlength** (Optional[float]) – Length of arrowhead [default: 0.3]
- **headwidth** (Optional[float]) – Width of arrowhead [default: 0.3]

at(*xy*)

Specify CurrentLabelInline position.

If xy is an Element, arrow will be placed along the element's leads and the arrow color will be inherited.

Parameters

- **xy** (XY | Element) – The absolute (x, y) position or an
- **on** (*Element instance to place the arrow*)

Return type
Element

class schemdraw.elements.lines.**DataBusLine**(**args*, ***kwargs*)

Straight Line with bus indication stripe

class schemdraw.elements.lines.**Dot**(***, *radius=None*, *open=None*, ***kwargs*)

Connection Dot

Keyword Arguments

- **radius** – Radius of dot [default: 0.075]

- **open** – Draw as an open circle [default: False]

class schemdraw.elements.lines.**DotDotDot**(*, *radius=None*, *open=None*, ***kwargs*)

Ellipsis element

Keyword Arguments

- **radius** – Radius of dots [default 0.075]
- **open** – Draw dots as open circles [default: False]

"Ellipsis" is a reserved keyword in Python used for slicing, thus the name DotDotDot.

class schemdraw.elements.lines.**Encircle**(*elm_list=None*, *, *padx=None*, *pady=None*, *includelabels=True*, ***kwargs*)

Draw ellipse around all elements in the list

Parameters

- **elm_list** (Optional[Sequence[*Element*]]) – List of elements to enclose
- **padx** (Optional[float]) – Horizontal distance from elements to loop [default: .2]
- **pady** (Optional[float]) – Vertical distance from elements to loop [default: .2]
- **includelabels** (bool) – Include labels in the ellipse

class schemdraw.elements.lines.**EncircleBox**(*elm_list=None*, *, *cornerradius=0.3*, *padx=None*, *pady=None*, *includelabels=True*, ***kwargs*)

Draw rounded box around all elements in the list

Parameters

- **elm_list** (Optional[Sequence[*Element*]]) – List elements to enclose
- **cornerraidus** – radius of corner rounding [default: 0.3]
- **padx** (Optional[float]) – Horizontal distance from elements to loop [default: 0.2]
- **pady** (Optional[float]) – Vertical distance from elements to loop [default: 0.2]
- **includelabels** (bool) – Include labels in the box

class schemdraw.elements.lines.**Gap**(**args*, ***kwargs*)

Gap for labeling port voltages, for example. Draws nothing, but provides place to attach a label such as ('+', 'V', '-').

Keyword Arguments

- **lblloc** – Label location within the gap [center]
- **lblalign** – Label alignment [(center, center)]
- **lblofst** – Offset to label [0]

class schemdraw.elements.lines.**Label**(*label=None*, ***kwargs*)

Label element.

For more options, use *Label().label()* method.

Parameters

label (Optional[str]) – text to display.

class schemdraw.elements.lines.**Line**(*, *arrow=None*, ***kwargs*)

Straight Line

Parameters

arrow (Optional[str]) – arrowhead specifier, such as '->', '<-', '<->', '-o', or '|->'

Keyword Arguments

- **arrowwidth** – Width of arrowhead [default: .15]
- **arrowlength** – Length of arrowhead [default: 0.25]

class schemdraw.elements.lines.**LoopArrow**(*direction='cw'*, *theta1=35*, *theta2=-35*, *width=1.0*, *height=1.0*, ***kwargs*)

Loop arrow, for mesh analysis notation

Parameters

- **direction** (Literal['cw', 'ccw']) – loop direction 'cw' or 'ccw'
- **theta1** (float) – Angle of start of loop arrow
- **theta2** (float) – Angle of end of loop arrow
- **width** (float) – Width of loop
- **height** (float) – Height of loop

class schemdraw.elements.lines.**LoopCurrent**(*elm_list=None*, *direction='cw'*, *theta1=35*, *theta2=-35*, *pad=0.2*, ***kwargs*)

Loop current label, for mesh analysis notation, placed within a box of 4 existing elements.

Parameters

- **elm_list** (Optional[Sequence[*Element*]]) – List of 4 elements surrounding loop, in order (top, right, bottom, left)
- **direction** (Literal['cw', 'ccw']) – loop direction 'cw' or 'ccw'
- **theta1** (float) – Angle of start of loop arrow
- **theta2** (float) – Angle of end of loop arrow
- **pad** (float) – Distance from elements to loop

class schemdraw.elements.lines.**Rect**(*corner1=(0, 0)*, *corner2=(1, 1)*, *, *fill=None*, *lw=None*, *ls=None*, ***kwargs*)

Rectangle Element

Used mainly for buliding more complex elements. Corner arguments are relative to Element coordinates, not Drawing coordinates.

Parameters

- **corner1** (Union[Tuple[float, float], Point]) – Position of top-left corner
- **corner2** (Union[Tuple[float, float], Point]) – Position of bottom-right corner
- **fill** (Optional[str]) – Color to fill if not None [default: inherit]
- **lw** (Optional[float]) – Line width [default: inherit]
- **ls** (Optional[Literal['-', ':', '--', '-.']]) – Line style '-', '–', ':', etc. [default: inherit]

class schemdraw.elements.lines.**Tag**(*, *width=None*, *height=0.625*, ***kwargs*)

Tag/flag element for labeling signal names.

Because text size is unknown until drawn, must specify width manually to fit a given text label.

Parameters

- **width** (Optional[float]) – Width of the tag [default: 1.5]
- **height** (Optional[float]) – Height of the tag [default: 0.625]

class schemdraw.elements.lines.**VoltageLabelArc**(*, *length=None*, *top=None*, *reverse=None*, *ofst=None*, *headlength=None*, *headwidth=None*, ***kwargs*)

Voltage Label as Arc along element path

Use *.at()* method to place the label over an existing element.

Parameters

- **top** (Optional[bool]) – Draw arrow on top or bottom of element [default: True]
- **reverse** (Optional[bool]) – Reverse the arrow direction
- **bend** – Distance to bend the arc at its maximum height
- **ofst** (Optional[float]) – Offset distance from centerline of element
- **length** (Optional[float]) – Length of the arrow [default: 1.75]
- **headlength** (Optional[float]) – Length of arrowhead [default: 0.3]
- **headwidth** (Optional[float]) – Width of arrowhead [default: 0.2]

class schemdraw.elements.lines.**Wire**(*shape='-'*, *k=1*, *, *arrow=None*, ***kwargs*)

Connect the .at() and .to() positions with lines depending on shape

Parameters

- **shape** (str) – Determines shape of wire: -: straight line |-: right-angle line starting vertically -|: right-angle line starting horizontally 'z': diagonal line with horizontal end segments 'N': diagonal line with vertical end segments *n*: n- or u-shaped lines *c*: c- or ⊃-shaped lines
- **k** (float) – Distance before the wire changes directions in *n* and *c* shapes.
- **arrow** (Optional[str]) – arrowhead specifier, such as '->', '<-', '<->', or '-o'

delta(*dx=0*, *dy=0*)

Specify ending position relative to start position

Return type
Element

dot(*open=False*)

Add a dot to the end of the element

Return type
Element

idot(*open=False*)

Add a dot to the input/start of the element

Return type
Element

to(*xy*, *dx=0*, *dy=0*)

Specify ending position

Parameters

- **xy** (Union[Tuple[float, float], Point]) – Ending position of element
- **dx** (float) – X-offset from xy position
- **dy** (float) – Y-offset from xy position

Return type
Element

class schemdraw.elements.lines.**ZLabel**(*, *ofst=None*, *hofst=None*, *length=None*, *lengthtip=None*, *headlength=None*, *headwidth=None*, ***kwargs*)

Right-angle arrow, often used to indicate impedance looking in to a node

Use *.at()* method to place the label over an existing element.

Parameters

- **ofst** (Optional[float]) – Vertical offset from centerline of element
- **hofst** (Optional[float]) – Horizontal offset from center of element
- **length** (Optional[float]) – Length of the arrow tail [default: 1]
- **lengthtip** (Optional[float]) – Length of the arrow tip [default: 0.5]
- **headlength** (Optional[float]) – Arrowhead length [default: 0.25]
- **headwidth** (Optional[float]) – Arrowhead width [default: 0.15]

at(*xy*)

Specify Element position.

If xy is an Element, arrow will be centered along element and its color will also be inherited.

Parameters

- **xy** (XY | Element) – The absolute (x, y) position or an
- **over** (*Element instance to center the arrow*)

Return type
Element

6.8.5 Cables and Connectors

Cable elements, coaxial and triaxial

class schemdraw.elements.cables.**Coax**(*, *length=None*, *radius=None*, *leadlen=None*, ***kwargs*)

Coaxial cable element.

Parameters

- **length** (Optional[float]) – Total length of the cable, excluding lead extensions. [default: 3]
- **radius** (Optional[float]) – Radius of shield [default: 0.3]
- **leadlen** (Optional[float]) – Distance (x) from start of center conductor to start of shield. [default: 0.6]

Anchors:

- shieldstart
- shieldstart_top
- shieldend
- shieldend_top
- shieldcenter
- shieldcenter_top

class schemdraw.elements.cables.**Triax**(*, *length=None*, *leadlen=None*, *radiusinner=None*, *radiusouter=None*, *shieldofststart=None*, *shieldofstend=None*, ***kwargs*)

Triaxial cable element.

Parameters

- **length** (Optional[float]) – Total length of the cable [default: 3]
- **radiusinner** (Optional[float]) – Radius of inner guard [default: 0.3]
- **radiusouter** (Optional[float]) – Radius of outer shield [default: 0.6]
- **leadlen** (Optional[float]) – Distance (x) from start of center conductor to start of guard. [default: 0.6]
- **shieldofststart** (Optional[float]) – Distance from start of inner guard to start of outer shield [default: 0.3]
- **shieldofstend** (Optional[float]) – Distance from end of outer shield to end of inner guard [default: 0.3]

Anchors:

- shieldstart
- shieldstart_top
- shieldend
- shieldend_top
- shieldcenter
- shieldcenter_top
- guardstart
- guardstart_top
- guardend
- guardend_top

Connectors and bus lines

class schemdraw.elements.connectors.**BusConnect**(*n=1*, *up=True*, *, *dy=None*, *lwbus=None*, *l=None*, ***kwargs*)

Data bus connection.

Adds the short diagonal lines that break out a bus (wide line) to connect to an Ic or Header element.

Parameters

- **n** (int) – Number of parallel lines
- **up** (bool) – Slant up or down
- **dy** (Optional[float]) – Distance between parallel lines [default: 0.6]
- **lwbus** (Optional[float]) – Line width of bus line [default: 4]
- **l** (Optional[float]) – length of connection lines [default: 3]

Anchors:

- start
- end
- p[X] where X is int for each data line

class schemdraw.elements.connectors.**BusLine**(*lw=None*, ***kwargs*)

Data bus line. Just a wide line.

Use BusConnect to break out connections to the BusLine.

Parameters

lw (Optional[float]) – Line width [default: 4]

class schemdraw.elements.connectors.**CoaxConnect**(*, *radius=None*, *radiusinner=None*, *fillinner=None*, ***kwargs*)

Coaxial connector

Parameters

- **radius** (Optional[float]) – Radius of outer shell [defualt: 0.4]
- **radiusinner** (Optional[float]) – Radius of inner conductor [default: 0.12]
- **fillinner** (Optional[str]) – Color to fill inner conductor [default: bg]

Anchors:

- center
- N
- S
- E
- W

class schemdraw.elements.connectors.**DA15**(*, *pinspacing=None*, *edge=None*, *number=None*, *pinfill=None*, ***kwargs*)

DA-15 Connector

Parameters

- **pinspacing** (Optional[float]) – Distance between pins [default: 0.6]
- **edge** (Optional[float]) – Distance between edge and pins [default: 0.3]
- **number** (Optional[bool]) – Draw pin numbers [default: False]
- **pinfill** (Optional[str]) – Color to fill pin circles [default: bg]

Anchors:

- pin1 thru pin15

class schemdraw.elements.connectors.**DB25**(*, *pinspacing=None*, *edge=None*, *number=None*, *pinfill=None*, ***kwargs*)

DB25 Connector

Parameters

- **pinspacing** (Optional[float]) – Distance between pins [default: 0.6]
- **edge** (Optional[float]) – Distance between edge and pins [default: 0.3]
- **number** (Optional[bool]) – Draw pin numbers
- **pinfill** (Optional[str]) – Color to fill pin circles [default: bg]

Anchors:

- pin1 thru pin25

schemdraw.elements.connectors.**DB9**

alias of *DE9*

class schemdraw.elements.connectors.**DC37**(*, *pinspacing=None*, *edge=None*, *number=None*, *pinfill=None*, ***kwargs*)

DC-37 Connector

Parameters

- **pinspacing** (Optional[float]) – Distance between pins [default: 0.6]
- **edge** (Optional[float]) – Distance between edge and pins [default: 0.3]
- **number** (Optional[bool]) – Draw pin numbers [default: False]
- **pinfill** (Optional[str]) – Color to fill pin circles [default: bg]

Anchors:

- pin1 thru pin37

class schemdraw.elements.connectors.**DD50**(*, *pinspacing=None*, *edge=None*, *number=None*, *pinfill=None*, ***kwargs*)

DD-50 Connector

Parameters

- **pinspacing** (Optional[float]) – Distance between pins [default: 0.6]
- **edge** (Optional[float]) – Distance between edge and pins [default: 0.3]
- **number** (Optional[bool]) – Draw pin numbers [default: False]
- **pinfill** (Optional[str]) – Color to fill pin circles [default: bg]

Anchors:

- pin1 thru pin50

class schemdraw.elements.connectors.**DE9**(*, *pinspacing=None*, *edge=None*, *number=None*, *pinfill=None*, ***kwargs*)

DE9 Connector (also known as DB9)

Parameters

- **pinspacing** (Optional[float]) – Distance between pins [default: 0.6]
- **edge** (Optional[float]) – Distance between edge and pins [default: 0.3]
- **number** (Optional[bool]) – Draw pin numbers [default: False]
- **pinfill** (Optional[str]) – Color to fill pin circles [default: bg]

Anchors:

- pin1 thru pin9

class schemdraw.elements.connectors.**Header**(*rows=4*, *cols=1*, *pinsleft=None*, *pinsright=None*, ***, *style=None*, *numbering=None*, *shownumber=None*, *pinalignleft=None*, *pinalignright=None*, *pinfontsizeright=None*, *pinfontsizeleft=None*, *pinspacing=None*, *edge=None*, *pinfill=None*, ***kwargs*)

Header connector element

Parameters

- **rows** (int) – Number of rows
- **cols** (int) – Number of columns. Pin numbering requires 1 or 2 columns
- **pinsleft** (Optional[Sequence[str]]) – List of pin labels for left side
- **pinsright** (Optional[Sequence[str]]) – List of pin labels for right side
- **style** (Optional[Literal['round', 'square', 'screw']]) – Connector style, 'round', 'square', or 'screw' [default: round]
- **numbering** (Optional[Literal['lr', 'ud', 'ccw']]) – Pin numbering order. 'lr' for left-to-right numbering, 'ud' for up-down numbering, or 'ccw' for counter-clockwise (integrated-circuit style) numbering. Pin 1 is always at the top-left corner, unless *flip* method is also called. [default: lr]
- **shownumber** (Optional[bool]) – Draw pin numbers outside the header
- **pinalignleft** (Optional[Literal['center', 'top', 'bottom', 'base']]) – Vertical alignment for pins on left side ('center', 'top', 'bottom') [default: 'bottom']
- **pinalignright** (Optional[Literal['center', 'top', 'bottom', 'base']]) – Vertical alignment for pins on right side ('center', 'top', 'bottom') [default: 'bottom']
- **pinfontsizeleft** (Optional[float]) – Font size for pin labels on left [default: 9]
- **pinfontsizeright** (Optional[float]) – Font size for pin labels on right [default: 9]
- **pinspacing** (Optional[float]) – Distance between pins [default: 0.6]
- **edge** (Optional[float]) – Distance between header edge and first pin row/column [default: 0.3]
- **pinfill** (Optional[str]) – Color to fill pin circles [default: bg]

Anchors:

pin[X] for each pin

class schemdraw.elements.connectors.**Jack**(**args*, ***kwargs*)

Jack (female connector)

class schemdraw.elements.connectors.**Jumper**(*, *pinspacing=None*, ***kwargs*)

Jumper for use on a Header element

Set position using *at()* method with a Header pin location, e.g. *Jumper().at(H.in1)*

Parameters
pinspacing (Optional[float]) – Spacing between pins [default: 0.6]

class schemdraw.elements.connectors.**OrthoLines**(*, *n=1*, *dy=None*, *xstart=None*, *arrow=None*, ***kwargs*)

Orthogonal multiline connectors

Use *at()* and *to()* methods to specify starting and ending location of OrthoLines.

The default lines are spaced to provide connection to pins with default spacing on Ic element or connector such as a Header.

Parameters

- **n** (int) – Number of parallel lines
- **dy** (Optional[float]) – Distance between parallel lines [default: 0.6]
- **xstart** (Optional[float]) – Fractional distance (0-1) to start vertical portion of first ortholine

delta(*dx=0*, *dy=0*)

Specify ending position relative to start position

Return type
Element

to(*xy*)

Specify ending position of OrthoLines

Return type
Element

class schemdraw.elements.connectors.**Plug**(**args*, ***kwargs*)

Plug (male connector)

class schemdraw.elements.connectors.**RightLines**(*, *n=1*, *dy=None*, *arrow=None*, ***kwargs*)

Right-angle multi-line connectors

Use *at()* and *to()* methods to specify starting and ending location.

The default lines are spaced to provide connection to pins with default spacing on Ic element or connector such as a Header.

Parameters

- **n** (int) – Number of parallel lines
- **dy** (Optional[float]) – Distance between parallel lines [default: 0.6]

delta(*dx=0*, *dy=0*)

Specify ending position relative to start position

Return type
Element

to(*xy*)

Specify ending position of OrthoLines

Return type
Element

class schemdraw.elements.connectors.**Terminal**(*, *r=None*, *open=None*, ***kwargs*)

Terminal element

Parameters

- **r** (Optional[float]) – Radius of terminal [default: 0.18]
- **open** (Optional[bool]) – Draw as open circle

6.8.6 Transistors

Transistor elements

class schemdraw.elements.transistors.**AnalogBiasedFet**(*, *bulk=None*, *offset_gate=None*, *source_arrow=None*, ***kwargs*)

Generic biased small-signal Field Effect Transistor, analog style

Parameters

- **bulk** (Optional[bool]) – Draw bulk contact
- **offset_gate** (Optional[bool]) – Draw gate on the source side of the transistor, rather than middle
- **source_arrow** (Optional[bool]) – Draw source dot on the transistor if bulk dot is not drawn

Anchors:
source drain gate bulk (if bulk=True) center

class schemdraw.elements.transistors.**AnalogNFet**(*, *bulk=None*, *offset_gate=None*, *source_arrow=None*, ***kwargs*)

N-type Field Effect Transistor, analog style

Parameters

- **bulk** (Optional[bool]) – Draw bulk contact
- **offset_gate** (Optional[bool]) – Draw gate on the source side of the transistor, rather than middle
- **source_arrow** (Optional[bool]) – Draw source arrow on the transistor if bulk arrow is not drawn

Anchors:
source drain gate bulk (if bulk=True) center

class schemdraw.elements.transistors.**AnalogPFet**(*, *bulk=None*, *offset_gate=None*, *source_arrow=None*, ***kwargs*)

P-type Field Effect Transistor, analog style

Parameters

- **bulk** (Optional[bool]) – Draw bulk contact

- **offset_gate** (Optional[bool]) – Draw gate on the source side of the transistor, rather than middle
- **source_arrow** (Optional[bool]) – Draw source arrow on the transistor if bulk arrow is not drawn

Anchors:
source drain gate bulk (if bulk=True) center

class schemdraw.elements.transistors.**Bjt**(*, *circle=None*, ***kwargs*)

Bipolar Junction Transistor (untyped)

Parameters
circle (Optional[bool]) – Draw circle around the transistor

Anchors:

- collector
- emitter
- base
- center

class schemdraw.elements.transistors.**Bjt2**(*, *circle=None*, ***kwargs*)

Bipolar Junction Transistor (untyped) which extends collector/emitter leads to the desired length

Parameters
circle (Optional[bool]) – Draw circle around the transistor

Anchors:

- collector
- emitter
- base

class schemdraw.elements.transistors.**BjtNpn**(*, *circle=None*, ***kwargs*)

NPN Bipolar Junction Transistor

Parameters
circle (Optional[bool]) – Draw circle around the transistor

Anchors:

- collector
- emitter
- base
- center

class schemdraw.elements.transistors.**BjtNpn2**(*, *circle=None*, ***kwargs*)

NPN Bipolar Junction Transistor which extends collector/emitter leads to the desired length

Parameters
circle (Optional[bool]) – Draw circle around the transistor

Anchors:

- collector
- emitter
- base

class schemdraw.elements.transistors.**BjtPnp**(*, *circle=None*, ***kwargs*)

PNP Bipolar Junction Transistor

Parameters
circle (Optional[bool]) – Draw circle around the transistor

Anchors:

- collector
- emitter
- base
- center

class schemdraw.elements.transistors.**BjtPnp2**(*, *circle=None*, ***kwargs*)

PNP Bipolar Junction Transistor which extends collector/emitter leads to the desired length

Parameters
circle (Optional[bool]) – Draw circle around the transistor

Anchors:

- collector
- emitter
- base

class schemdraw.elements.transistors.**BjtPnp2c**(*, *circle=None*, ***kwargs*)

PNP Bipolar Junction Transistor with 2 collectors

Parameters
circle (Optional[bool]) – Draw circle around the transistor

Anchors:

- collector
- emitter
- base
- C2
- center

class schemdraw.elements.transistors.**BjtPnp2c2**(*, *circle=None*, ***kwargs*)

2-Collector PNP Bipolar Junction Transistor which extends collector/emitter leads to the desired length

Parameters
circle (Optional[bool]) – Draw circle around the transistor

Anchors:

- collector
- emitter
- base
- C2

class schemdraw.elements.transistors.**JFet**(*, *circle=None*, ***kwargs*)

Junction Field Effect Transistor (untyped)

Anchors:

- source
- drain
- gate
- center

class schemdraw.elements.transistors.**JFet2**(*, *circle=None*, ***kwargs*)

Junction Field Effect Transistor (untyped) which extends collector/emitter leads to the desired length

Anchors:

- source
- drain
- gate

class schemdraw.elements.transistors.**JFetN**(*, *circle=None*, ***kwargs*)

N-type Junction Field Effect Transistor

Parameters
circle (Optional[bool]) – Draw circle around the transistor

Anchors:

- source
- drain
- gate
- center

class schemdraw.elements.transistors.**JFetN2**(*, *circle=None*, ***kwargs*)

N-type Junction Field Effect Transistor which extends collector/emitter leads to the desired length

Parameters
circle (Optional[bool]) – Draw circle around the transistor

Anchors:

- source
- drain
- gate

class schemdraw.elements.transistors.**JFetP**(*, *circle=None*, ***kwargs*)

P-type Junction Field Effect Transistor

Parameters
circle (Optional[bool]) – Draw circle around the transistor

Anchors:

- source
- drain
- gate
- center

class schemdraw.elements.transistors.**JFetP2**(*, *circle=None*, ***kwargs*)

P-type Junction Field Effect Transistor which extends collector/emitter leads to the desired length

Parameters
circle (Optional[bool]) – Draw circle around the transistor

Anchors:

- source
- drain
- gate

class schemdraw.elements.transistors.**NFet**(*, *bulk=None*, ***kwargs*)

N-type Field Effect Transistor

Parameters
bulk (Optional[bool]) – Draw bulk contact [default: False]

Anchors:

- source
- drain
- gate
- center

class schemdraw.elements.transistors.**NFet2**(*, *bulk=None*, ***kwargs*)

N-type Field Effect Transistor which extends source/drain leads to the desired length

Parameters
bulk (Optional[bool]) – Draw bulk contact [default: False]

Anchors:

- source
- drain
- gate

class schemdraw.elements.transistors.**NMos**(*diode=False*, *circle=False*, ***kwargs*)

N-type Metal Oxide Semiconductor Field Effect Transistor

Args:
diode: Draw body diode circle: Draw circle around the mosfet

Anchors:

- source
- drain
- gate

Note: vertical orientation. For horizontal orientation, see NMos2.

class schemdraw.elements.transistors.**NMos2**(*diode=False*, *circle=False*, ***kwargs*)

N-type Metal Oxide Semiconductor Field Effect Transistor

Args:
diode: Draw body diode circle: Draw circle around the mosfet circle_lw: Line width of circle

Anchors:

- source
- drain
- gate

Note: horizontal orientation. For vertical orientation, see NMos.

class schemdraw.elements.transistors.**PFet**(*, *bulk=None*, ***kwargs*)

P-type Field Effect Transistor

Parameters
bulk (Optional[bool]) – Draw bulk contact

Anchors:
source drain gate center

class schemdraw.elements.transistors.**PFet2**(*, *bulk=None*, ***kwargs*)

P-type Field Effect Transistor which extends source/drain leads to the desired length

Parameters
bulk (Optional[bool]) – Draw bulk contact [default: False]

Anchors:

- source
- drain
- gate

class schemdraw.elements.transistors.**PMos**(*diode=False*, *circle=False*, ***kwargs*)

P-type Metal Oxide Semiconductor Field Effect Transistor

Args:
diode: Draw body diode circle: Draw circle around the mosfet

Anchors:

- source

- drain
- gate

Note: vertical orientation. For horizontal orientation, see PMos2.

class schemdraw.elements.transistors.**PMos2**(*diode=False*, *circle=False*, ***kwargs*)

P-type Metal Oxide Semiconductor Field Effect Transistor

Args:
diode: Draw body diode circle: Draw circle around the mosfet

Anchors:

- source
- drain
- gate

Note: horizontal orientation. For vertical orientation, see PMos.

6.8.7 Transformers

Transformer element definitions

class schemdraw.elements.xform.**Transformer**(*t1=4*, *t2=4*, *, *core=None*, *loop=None*, *align='center'*, ***kwargs*)

Add taps to the windings on either side using the *.taps* method.

Parameters

- **t1** (int | Sequence[int]) – Turns on primary (left) side
- **t2** (int | Sequence[int]) – Turns on secondary (right) side
- **core** (Optional[bool]) – Draw the core (parallel lines) [default: True]
- **loop** (Optional[bool]) – Use spiral/cycloid (loopy) style [default: False]

Anchors:

- p1: primary side 1
- p2: primary side 2
- s1: secondary side 1
- s2: secondary side 2
- Other anchors defined by *taps* method

tap(*name*, *pos*, *side='primary'*)

Add a tap

A tap is simply a named anchor definition along one side of the transformer.

Parameters

- **name** (str) – Name of the tap/anchor
- **pos** (int) – Turn number from the top of the tap
- **side** (Literal['primary', 'secondary', 'left', 'right']) – Primary (left) or Secondary (right) side

Return type
Transformer

6.8.8 Opamp and Integrated Circuits

Operation amplifier

class schemdraw.elements.opamp.**Opamp**(*, *sign=None*, *leads=None*, ***kwargs*)

Operational Amplifier.

Keyword Arguments

- **sign** – Draw +/- labels at each input
- **leads** – Draw short leads on input/output

Anchors:

- in1
- in2
- out
- vd
- vs
- n1
- n2
- n1a
- n2a

Integrated Circuit Element

class schemdraw.elements.intcircuits.**DFlipFlop**(*preclr=False*, *preclrinvert=True*, ***kwargs*)

D-Type Flip Flop

Parameters

- **preclr** (bool) – Show preset and clear inputs
- **preclrinvert** (bool) – Add invert bubble to preset and clear inputs

Keyword Arguments
size – Size of the box [default: (2, 3)]

Anchors:

- D
- CLK
- Q
- Qbar
- PRE
- CLR

class schemdraw.elements.intcircuits.**Ic**(*size=None*, *pins=None*, *slant=0*, ***kwargs*)

Integrated Circuit - box with arbitrary pins and labels along each side

Parameters

- **size** (Union[Tuple[float, float], Point, None]) – (width, height) in drawing units
- **pins** (Optional[Sequence[*IcPin*]]) – list of IcPin instances
- **slant** (float) – Angle to slant top/bottom of IC (ie for multiplexers)
- **pinspacing** – spacing between pins, in drawing units

pin(*side='L'*, *name=None*, *pin=None*, *pos=None*, *slot=None*, *invert=False*, *invertradius=0.15*, *color=None*, *rotation=0*, *anchorname=None*, *lblsize=None*, *href=None*, *decoration=None*)

Add a pin to the IC

Parameters

- **name** (str | None) – Input/output name (inside the box)
- **pin** (str | None) – Pin name/number (outside the box)
- **side** (Side) – Side of box for the pin, 'left', 'right', 'top', 'bottom'
- **pos** (float | None) – Pin position along the side, fraction from 0-1
- **slot** (str | None) – Slot definition of pin location, given in 'X/Y' format. '2/4' is the second pin on a side with 4 pins.
- **invert** (bool) – Add an invert bubble to the pin
- **invertradius** (float) – Radius of invert bubble
- **color** (str | None) – Color for the pin and label
- **rotation** (float) – Rotation for label text
- **anchorname** (str | None) – Named anchor for the pin
- **lblsize** (float | None) – Font size for label
- **href** (Optional[str]) – Hyperline target (jump to)
- **decoration** (Optional[str]) – "underline" or "overline"

property **pinnames: list[str]**

List of all pin names

side(*side='L'*, *spacing=0*, *pad=0.5*, *leadlen=0.5*, *label_ofst=0.15*, *label_size=14.0*, *pinlabel_ofst=0.05*, *pinlabel_size=11.0*)

Set parameters for spacing/layout of one side

Parameters

- **side** (Literal['top', 'bot', 'lft', 'rgt', 'bottom', 'left', 'right', 'L', 'R', 'T', 'B']) – Side of box to define
- **spacing** (float) – Distance between pins
- **pad** (float) – Distance from box edge to first pin
- **leadlen** (float) – Length of pin lead extensions
- **label_ofst** (float) – Offset between box and label (inside box)
- **label_size** (float) – Font size of label (inside box)

- **pinlabel_ofst** (float) – Offset between box and pin label (outside box)
- **pinlabel_size** (float) – Font size of pin label (outside box)

class schemdraw.elements.intcircuits.**Ic555**(**args*, ***kwargs*)

555 Timer IC

class schemdraw.elements.intcircuits.**IcBox**(*w*, *h*, *y1*, *y2*)

h

Alias for field number 1

w

Alias for field number 0

y1

Alias for field number 2

y2

Alias for field number 3

class schemdraw.elements.intcircuits.**IcDIP**(**, pins=None, names=None, notch=None, width=None, pinw=None, spacing=None, number=None, fontsize=None, pfontsize=None, **kwargs*)

Dual-inline Package Integrated Circuit.

Parameters

names (Optional[Sequence[str]]) – List of names for each pin to display inside the box

Keyword Arguments

- **pins** – number of pins [default: 8]
- **notch** – Show the notch at top of box [default: True]
- **width** – Width of the box [default: 3]
- **pinw** – Width and height of each pin [default: 0.6]
- **spacing** – Distance between each pin [default: 0.5]
- **number** – Show pin numbers inside each pin [default: True]
- **fontsize** – Size for pin name labels [default: 12]
- **pfontsize** – Size for pin number labels [default: 10]

Anchors:

- p[x] - Each pin
- p[x]_in - Inside contact for each pin

If signal names are provided, they will also be added as anchors along with _in inside variants.

class schemdraw.elements.intcircuits.**IcPin**(*name=None, pin=None, side='L', pos=None, slot=None, invert=False, invertradius=0.15, color=None, rotation=0, anchorname=None, lblsize=None, href=None, decoration=None, pinlblsize=None*)

Integrated Circuit Pin

Parameters

- **name** (str | None) – Input/output name (inside the box)
- **pin** (str | None) – Pin name/number (outside the box)
- **side** (Side) – Side of box for the pin, 'left', 'right', 'top', 'bottom'
- **pos** (float | None) – Pin position along the side, fraction from 0-1
- **slot** (str | None) – Slot definition of pin location, given in 'X/Y' format. '2/4' is the second pin on a side with 4 pins.
- **invert** (bool) – Add an invert bubble to the pin
- **invertradius** (float) – Radius of invert bubble
- **color** (str | None) – Color for the pin and label
- **rotation** (float) – Rotation for label text
- **anchorname** (str | None) – Named anchor for the pin

class schemdraw.elements.intcircuits.**IcSide**(*spacing=0.0*, *pad=0.25*, *leadlen=0.5*, *label_ofst=0.15*, *label_size=14.0*, *pinlabel_ofst=0.05*, *pinlabel_size=11.0*)

Pin layout parameters for one side of an Ic

class schemdraw.elements.intcircuits.**JKFlipFlop**(*preclr=False*, *preclrinvert=True*, ***kwargs*)

J-K Flip Flop

Parameters

- **preclr** (bool) – Show preset and clear inputs
- **preclrinvert** (bool) – Add invert bubble to preset and clear inputs

Keyword Arguments

size – Size of the box [default: (2, 3)]

Anchors:

- J
- K
- CLK
- Q
- Qbar
- PRE
- CLR

class schemdraw.elements.intcircuits.**Multiplexer**(*demux=False*, *size=None*, *pins=None*, *slant=25*, ***kwargs*)

Parameters

- **demux** (bool) – Draw as demultiplexer
- **size** (Union[Tuple[float, float], Point, None]) – (Width, Height) of box
- **pins** (Optional[Sequence[*IcPin*]]) – List of IcPin instances defining the inputs/outputs
- **slant** (float) – Slant angle of top/bottom edges

Keyword Arguments

- **pinspacing** – Spacing between pins [default: 0.6]
- **edgepadH** – Padding between top/bottom and first pin [default: 0.25]
- **edgepadW** – Padding between left/right and first pin [default: 0.25]
- **lofst** – Offset between edge and label (inside box) [default: 0.15]
- **lsize** – Font size of labels (inside box) [default: 14]
- **plblofst** – Offset between edge and pin label (outside box) [default: 0.05]
- **plblsize** – Font size of pin labels (outside box) [default: 11]

If a pin is named '>', it will be drawn as a proper clock signal input.

Anchors:

- inL[X] - Each pin on left side
- inR[X] - Each pin on right side
- inT[X] - Each pin on top side
- inB[X] - Each pin on bottom side
- pin[X] - Each pin with a number
- CLK (if clock pin is defined with '>' name)

Pins with names are also defined as anchors (if the name does not conflict with other attributes).

class schemdraw.elements.intcircuits.**SevenSegment**(*, *decimal=None*, *digit=None*, *segcolor=None*, *tilt=None*, *labelsegments=None*, *anode=None*, *cathode=None*, ***kwargs*)

A seven-segment display digit.

Keyword Arguments

- **decimal** – Show decimal point segment [default: False]
- **digit** – Number to display [default: 8]
- **segcolor** – Color of segments [default: red]
- **tilt** – Tilt angle in degrees [default: 10]
- **labelsegments** – Add a-g labels to each segment [default: True]
- **anode** – Add common anode pin [default: False]
- **cathode** – Add common cathode pin [default: False]

Anchors:

- a
- b
- c
- d
- e
- f
- g
- dp

- cathode
- anode

class schemdraw.elements.intcircuits.**VoltageRegulator**(**args*, ***kwargs*)

Voltage regulator

Keyword Arguments
size – Size of the box [default: (2, 1.5)]

Anchors:

- in
- out
- gnd

schemdraw.elements.intcircuits.**sevensegdigit**(*bottom=0*, *left=0*, *seglen=1.5*, *segw=0.3*, *spacing=0.12*, *decimal=False*, *digit=8*, *segcolor='red'*, *tilt=10*, *labelsegments=True*)

Generate drawing segments for a 7-segment display digit. Use for building new elements incorporating a 7-segment display.

Parameters

- **bottom** (float) – Location of bottom of digit
- **left** (float) – Location of left side of digit
- **seglen** (float) – Length of one segment
- **segw** (float) – Width of one segment
- **spacing** (float) – Distance between segments in corners
- **decimal** (bool) – Show decimal point segment
- **digit** (int | str) – Number to display
- **segcolor** (str) – Color of segments
- **tilt** (float) – Tilt angle in degrees
- **labelsegments** (bool) – Add a-g labels to each segment
- **anode** – Add common anode pin
- **cathode** – Add common cathode pin
- **size** – Total size of the box

Return type
list[SegmentType]

Returns
List of Segments making the digit

6.8.9 Other

Other elements

class schemdraw.elements.misc.**AudioJack**(*, *radius=None*, *ring=False*, *ringswitch=False*, *dots=True*, *switch=False*, *open=None*, ***kwargs*)

Audio Jack with 2 or 3 connectors and optional switches.

Parameters

- **ring** (bool) – Show ring (third conductor) contact
- **switch** (bool) – Show switch on tip contact
- **ringswitch** (bool) – Show switch on ring contact
- **dots** (bool) – Show connector dots
- **radius** (Optional[float]) – Radius of connector dots [default: 0.075]

Anchors:

- tip
- sleeve
- ring
- ringswitch
- tipswitch

class schemdraw.elements.misc.**Mic**(**args*, ***kwargs*)

Microphone element with two inputs.

Anchors:

- in1
- in2

class schemdraw.elements.misc.**Motor**(**args*, ***kwargs*)

class schemdraw.elements.misc.**Speaker**(**args*, ***kwargs*)

Speaker element with two inputs.

Anchors:

- in1
- in2

Compound elements made from groups of other elements

class schemdraw.elements.compound.**ElementCompound**(***kwargs*)

Element onto which other elements can be added like a drawing

add(*element*)

Add an element to the segments list

Return type
Element

move(*dx=0*, *dy=0*)

Move relative to current position

Return type
None

move_from(*xy*, *dx=0*, *dy=0*, *theta=None*)

Move relative to xy position

Return type
None

class schemdraw.elements.compound.**Optocoupler**(*box=True*, *boxfill='none'*, *boxpad=0.2*, *base=False*, ***kwargs*)

Optocoupler element

Parameters

- **box** (bool) – Draw a box around the optocoupler
- **boxfill** (str) – Color to fill the box
- **boxpad** (float) – Padding between phototransistor and box
- **base** (bool) – Add a base contact to the phototransistor

Anchors:

- anode
- cathode
- emitter
- collector
- base (if base==True)

class schemdraw.elements.compound.**Rectifier**(**args*, ***kwargs*)

Diode Rectifier Bridge

Parameters

- **fill** – Fill the didoes
- **labels** – Labels to draw on each resistor

Anchors:

- N
- S
- E
- W

class schemdraw.elements.compound.**Relay**(*unit=2*, *cycl=False*, *switch='spst'*, *core=True*, *box=True*, *boxfill='none'*, *boxpad=0.25*, *swreverse=False*, *swflip=False*, *link=True*, ***kwargs*)

Relay element with an inductor and switch

Parameters

- **unit** (float) – Unit length of the inductor
- **cycloid** – Use cycloid style inductor
- **switch** (str) – Switch style 'spst', 'spdt', 'dpst', 'dpdt'
- **swreverse** (bool) – Reverse the switch
- **swflip** (bool) – Flip the switch up/down
- **core** (bool) – Show inductor core bar
- **link** (bool) – Show dotted line linking inductor and switch
- **box** (bool) – Draw a box around the relay
- **boxfill** (str) – Color to fill the box
- **boxpad** (float) – Spacing between components and box

class schemdraw.elements.compound.**Wheatstone**(*vout=False*, *labels=None*, ***kwargs*)

Wheatstone Resistor Bridge

Parameters

- **vout** (bool) – draw output terminals inside the bridge
- **labels** (Optional[Sequence[str]]) – Labels to draw on each resistor

Anchors:

- N
- S
- E
- W
- vo1 (if vout==True)
- vo2 (if vout==True)

Twoport elements made from groups of other elements

class schemdraw.elements.twoports.**CurrentTransactor**(*reverse_output=False*, ***kwargs*)

Current transactor

Parameters

- **bpadx** – Horizontal padding from edge of either component
- **bpady** – Vertical padding from edge of either component
- **minw** – Margin around component if smaller than minw
- **terminals** – Draw with terminals extending past box
- **component_offset** – Offset between input and output element
- **box** – Draw twoport outline
- **boxfill** – Color to fill the twoport if not None
- **boxlw** – Line width of twoport outline
- **boxls** – Line style of twoport outline '-', '–', ':', etc.
- **reverse_output** (bool) – Switch direction of output source, defaults to False

Anchors:

- in_p
- in_n
- out_p
- out_n
- center

class schemdraw.elements.twoports.**ElementTwoport**(*input_element*, *output_element*, *boxpadx=0.2*, *boxpady=0.2*, *minw=0.5*, *terminals=True*, *unit=1.5*, *width=2.15*, *box=True*, *boxfill=None*, *boxlw=None*, *boxls=None*, ***kwargs*)

Compound twoport element

Parameters

- **input_element** (type[*Element2Term*]) – The element forming the input branch of the twoport
- **output_element** (type[*Element2Term*]) – The element forming the output branch of the twoport
- **bpadx** – Horizontal padding from edge of either component
- **bpady** – Vertical padding from edge of either component
- **minw** (float) – Margin around component if smaller than minw
- **terminals** (bool) – Draw with terminals extending past box
- **unit** (float) – Length of input and output element
- **width** (float) – Width of the twoport box
- **box** (bool) – Draw twoport outline
- **boxfill** (Optional[str]) – Color to fill the twoport if not None
- **boxlw** (Optional[float]) – Line width of twoport outline
- **boxls** (Optional[Literal['-', ':', '--', '-.']]) – Line style of twoport outline '-', '–', ':', etc.

Anchors:

- in_p
- in_n
- out_p
- out_n
- center

class schemdraw.elements.twoports.**Nullor**(**args*, ***kwargs*)

Parameters

- **bpadx** – Horizontal padding from edge of either component
- **bpady** – Vertical padding from edge of either component

- **minw** – Margin around component if smaller than minw
- **terminals** – Draw with terminals extending past box
- **component_offset** – Offset between input and output element
- **box** – Draw twoport outline
- **boxfill** – Color to fill the twoport if not None
- **boxlw** – Line width of twoport outline
- **boxls** – Line style of twoport outline '-', '–', ':', etc.

Anchors:

- in_p
- in_n
- out_p
- out_n
- center

class schemdraw.elements.twoports.**TransadmittanceTransactor**(*reverse_output=False*, ***kwargs*)

Transadmittance transactor

Parameters

- **bpadx** – Horizontal padding from edge of either component
- **bpady** – Vertical padding from edge of either component
- **minw** – Margin around component if smaller than minw
- **terminals** – Draw with terminals extending past box
- **component_offset** – Offset between input and output element
- **box** – Draw twoport outline
- **boxfill** – Color to fill the twoport if not None
- **boxlw** – Line width of twoport outline
- **boxls** – Line style of twoport outline '-', '–', ':', etc.
- **reverse_output** (bool) – Switch direction of output source, defaults to False

Anchors:

- in_p
- in_n
- out_p
- out_n
- center

class schemdraw.elements.twoports.**TransimpedanceTransactor**(*reverse_output=False*, ***kwargs*)

Transimpedance transactor

Parameters

- **bpadx** – Horizontal padding from edge of either component
- **bpady** – Vertical padding from edge of either component
- **minw** – Margin around component if smaller than minw
- **terminals** – Draw with terminals extending past box
- **component_offset** – Offset between input and output element
- **box** – Draw twoport outline
- **boxfill** – Color to fill the twoport if not None
- **boxlw** – Line width of twoport outline
- **boxls** – Line style of twoport outline '-', '–', ':', etc.
- **reverse_output** (bool) – Switch direction of output source, defaults to False

Anchors:

- in_p
- in_n
- out_p
- out_n
- center

class schemdraw.elements.twoports.**TwoPort**(*sign=True*, *arrow=True*, *reverse_output=False*, ***kwargs*)

Generic Twoport

Parameters

- **bpadx** – Horizontal padding from edge of either component
- **bpady** – Vertical padding from edge of either component
- **minw** – Margin around component if smaller than minw
- **terminals** – Draw with terminals extending past box
- **component_offset** – Offset between input and output element
- **box** – Draw twoport outline
- **boxfill** – Color to fill the twoport if not None
- **boxlw** – Line width of twoport outline
- **boxls** – Line style of twoport outline '-', '–', ':', etc.
- **sign** (bool) – Draw input and output terminal labels
- **arrow** (bool) – Draw arrow from input to output

Anchors:

- in_p

- in_n
- out_p
- out_n
- center

class schemdraw.elements.twoports.**VMCMPair**(**args*, ***kwargs*)

Nullor

Parameters

- **bpadx** – Horizontal padding from edge of either component
- **bpady** – Vertical padding from edge of either component
- **minw** – Margin around component if smaller than minw
- **terminals** – Draw with terminals extending past box
- **component_offset** – Offset between input and output element
- **box** – Draw twoport outline
- **boxfill** – Color to fill the twoport if not None
- **boxlw** – Line width of twoport outline
- **boxls** – Line style of twoport outline '-', '–', ':', etc.

Anchors:

- in_p
- in_n
- out_p
- out_n
- center

class schemdraw.elements.twoports.**VoltageTransactor**(*reverse_output=False*, ***kwargs*)

Voltage transactor

Parameters

- **bpadx** – Horizontal padding from edge of either component
- **bpady** – Vertical padding from edge of either component
- **minw** – Margin around component if smaller than minw
- **terminals** – Draw with terminals extending past box
- **component_offset** – Offset between input and output element
- **box** – Draw twoport outline
- **boxfill** – Color to fill the twoport if not None
- **boxlw** – Line width of twoport outline
- **boxls** – Line style of twoport outline '-', '–', ':', etc.
- **reverse_output** (bool) – Switch direction of output source, defaults to False

Anchors:

- in_p
- in_n
- out_p
- out_n
- center

Vacuum Tubes

class schemdraw.elements.tubes.**DualVacuumTube**(*cathode='heated'*, *anodetype='plate'*, *grids_left=1*, *grids_right=1*, *heater=True*, ***kwargs*)

Dual Vacuum Tube

Keyword Arguments

- **cathode** – Cathode style 'heated', 'cold', or 'none'
- **anodetype** – Anode style 'plate', 'dot', or 'none'
- **grids_left** – Number of grids for left element
- **grids_right** – Number of grids for right element
- **heater** – Show heater filament

Anchors:

- anodeA
- anodeB
- gridA[_X]
- gridA_[X]R
- gridB[_X]
- gridB_[X]R
- cathodeA
- cathodeA_R
- cathodeB
- cathodeB_R
- heat1
- heat2

class schemdraw.elements.tubes.**NixieTube**(*anodes=3*, *cathode='T'*, *anodetype='narrow'*, *grid=False*, ***kwargs*)

Nixie Tube

Keyword Arguments

- **anodes** – Number of anode connections
- **cathode** – Cathode style 'T', 'cold', or 'none'
- **anodetype** – Anode type 'narrow' or 'dot'

- **grid** – Show grid

Anchors:

- anode[X]
- cathode
- grid
- grid_R

class schemdraw.elements.tubes.**Pentode**(*strap=False*, *heater=False*, *split='none'*, ***kwargs*)

Pentode Vacuum Tube

Keyword Arguments

- **strap** – Connect suppressor grid to cathode
- **heater** – Show heater filament
- **split** – Draw open envelope on 'left' or 'right' side

Anchors:

- anode
- cathode
- cathode_R
- suppressor
- suppressor_R
- screen
- screen_R
- control
- control_R
- heat1
- heat2

class schemdraw.elements.tubes.**Tetrode**(*heater=False*, *split='none'*, ***kwargs*)

Tetrode Vacuum Tube

Keyword Arguments

- **heater** – Show heater filament
- **split** – Draw open envelope on 'left' or 'right' side

Anchors:

- anode
- cathode
- cathode_R
- screen
- screen_R

- control
- control_R
- heat1
- heat2

class schemdraw.elements.tubes.**Triode**(*heater=False*, *split='none'*, ***kwargs*)

Triode Vacuum Tube

Keyword Arguments

- **heater** – Show heater filament
- **split** – Draw open envelope on 'left' or 'right' side

Anchors:

- anode
- cathode
- cathode_R
- control
- control_R
- heat1
- heat2

class schemdraw.elements.tubes.**TubeBase**(***kwargs*)

Base class for vacuum tubes

class schemdraw.elements.tubes.**TubeDiode**(*heater=False*, *split='none'*, ***kwargs*)

Vacuum Tube Diode

Keyword Arguments

- **heater** – Show heater filament
- **split** – Draw open envelope on 'left' or 'right' side

Anchors:

- anode
- cathode
- cathode_R
- heat1
- heat2

class schemdraw.elements.tubes.**VacuumTube**(*, *cathode='heated'*, *anodetype='plate'*, *grids=1*, *heater=True*, *split='none'*, ***kwargs*)

Generic Configrable Vacuum Tube

Keyword Arguments

- **cathode** – Cathode style 'heated', 'cold', or 'none'

- **anodetype** – Anode style 'plate', 'dot', 'narrow', or 'none'
- **grids** – Number of grids
- **heater** – Show heater filament
- **split** – Draw open envelope on 'left' or 'right' side

Style Parameters:
tube_lw: linewidth of envelope anode_lw: linewidth of anode cathode_lw: linewidth of cathode heat_lw: linewidth of heater grid_lw' linewidth of grid n_grid_dashes: number of grid dashes dot_radius: radius of dot for cold cathode

Anchors:

- anode
- grid[_X]
- grid_[X]R
- cathode
- cathode_R
- heat1
- heat2

6.9 Logic Gates

Logic gate definitions

class schemdraw.logic.logic.**And**(*inputs=2*, *nand=False*, *inputnots=None*, ***, *leadin=None*, *leadout=None*, ***kwargs*)

AND gate

Parameters

- **inputs** (int) – Number of inputs to gate.
- **nand** (bool) – Draw invert bubble on output
- **inputnots** (Optional[Sequence[int]]) – Input numbers (starting at 1) of inputs that have invert bubble
- **leadin** (Optional[float]) – Length of input leads [default: 0.35]
- **leadout** (Optional[float]) – Length of output lead [default: 0.35]

Anchors:
out in[X] - for each input

class schemdraw.logic.logic.**Buf**(**args*, ***kwargs*)

Buffer

Anchors:
in out

class schemdraw.logic.logic.**Not**(**args*, ***kwargs*)

Not gate/inverter

Anchors:
in out

class schemdraw.logic.logic.**NotNot**(**args*, ***kwargs*)

Double inverter

Anchors:
in out

class schemdraw.logic.logic.**Or**(*inputs=2*, *nor=False*, *xor=False*, *inputnots=None*, ***, *leadin=0.35*, *leadout=0.35*, ***kwargs*)

OR or XOR gate element.

Parameters

- **inputs** (int) – Number of inputs to gate.
- **nor** (bool) – Draw invert bubble on output
- **xor** (bool) – Draw as exclusive-or gate
- **inputnots** (Optional[Sequence[int]]) – Input numbers (starting at 1) of inputs that have invert bubble
- **leadin** (float) – Length of input leads [default: 0.35]
- **leadout** (float) – Length of output lead [default: 0.35]

Anchors:
out in[X] - for each input

class schemdraw.logic.logic.**Schmitt**(**args*, ***kwargs*)

Schmitt Trigger

Anchors:
in out

class schemdraw.logic.logic.**SchmittAnd**(***, *leadin=0.35*, *leadout=0.35*, ***kwargs*)

Schmitt Trigger AND

Anchors:
in1 in2 out

class schemdraw.logic.logic.**SchmittNot**(**args*, ***kwargs*)

Inverted Schmitt Trigger

Anchors:
in out

class schemdraw.logic.logic.**Tgate**(**args*, ***kwargs*)

Transmission gate.

Anchors:
in out c cbar

class schemdraw.logic.logic.**Tristate**(*outputnot=True*, *controlnot=False*, ***kwargs*)

Tristate inverter :type outputnot: bool :param outputnot: Draw invert bubble on output :type controlnot: bool :param controlnot: Active-low control

Anchors:
in out c

logic_parser.**logicparse**(*gateW=2*, *gateH=0.75*, *outlabel=None*)

Parse a logic string expression and draw the gates in a schemdraw Drawing

Logic expression is defined by string using 'and', 'or', 'not', etc. for example, "a or (b and c)". Parser recognizes several symbols and names for logic functions: [and, '&', '∧'] [or, '|', '∨', '+'] [xor, '⊕', '⊻'] [not, '~', '¬']

Parameters

- **expr** (str) – Logic expression
- **gateH** (float) – Height of one gate
- **gateW** (float) – Width of one gate
- **outlabel** (Optional[str]) – Label for logic output

Return type
Drawing

Returns
schemdraw.Drawing with logic tree

class schemdraw.logic.table.**Table**(*table*, *colfmt=None*, *fontsize=12*, *font='sans'*, ***kwargs*)

Table Element for drawing rudimentary Markdown formatted tables, such as logic truth tables.

Parameters

- **table** (str) – Table definition, as markdown string. Columns separated by |. Separator rows contain — or === between column separators.
- **colfmt** (Optional[str]) – Justification and vertical separators to draw for each column, similar to LaTeX tabular environment parameter. Justification characters include 'c', 'r', and 'l' for center, left, and right justification. Separator characters may be '|' for a single vertical bar, or '‖' or '||' for a double vertical bar, or omitted for no bar. Example: 'cc|c'.
- **fontsize** (float) – Point size of table font
- **font** (str) – Name of table font

Example Table:
| A | B | Y | |—|—|—| | 0 | 0 | 1 | | 0 | 1 | 0 | | 1 | 0 | 0 | | 1 | 1 | 0 |

class schemdraw.logic.kmap.**Kmap**(*names='ABCD'*, *truthtable=None*, *groups=None*, *default='0'*, ***kwargs*)

Karnaugh Map

Draws a K-Map with 2, 3, or 4 variables.

Parameters

- **names** (str) – 2, 3, or 4-character string defining names of the inputs
- **truthtable** (Optional[Sequence[Sequence[Union[int, str]]]]) – list defining values to display in each box of the K-Map. First element is string of 2, 3, or 4 logic 0's and 1's, and last element is the string to display for that input. Example: ('0000', '1') displays a '1' when all inputs are 0.
- **groups** (Optional[dict]) – dictionary of style parameters for circling groups of inputs. Dictionary key must be same length as names, and defines which elements are circled using '0', '1', or '.' in each position. For example, '1…' circles every box where A=1, and '.11.'

circles every box where both B and C are 1. Value of dictionary pair is another dictionary containing style of box (e.g. color, fill, lw, and ls).

- **default** (str) – string to display in boxes that don't have a truthtable entry defined

Anchors:

- cellXXXX - Center of each cell in the grid, where X is 0 or 1

class schemdraw.logic.timing.**TimingDiagram**(*waved*, ***kwargs*)

Logic Timing Diagram

Draw timing diagrams compatible with WaveJSON format See https://wavedrom.com/ for details. Use *from_json* to use WaveJSON strings copied from the site (since they can't be copied as proper Python dicts due to lack of quoting).

Schemdraw provides a few additional extensions to the WaveJSON dictionary, including asynchronous waveforms and configuration options (color, lw) on each wave. See documentation for full specification.

Parameters
wave – WaveJSON as a Python dict

Keyword Arguments

- **yheight** – Height of one waveform
- **ygap** – Separation between two waveforms
- **risetime** – Rise/fall time for wave transitions
- **fontsize** – Size of label fonts
- **datafontsize** – Size of data font
- **nodesize** – Size of node labels
- **namecolor** – Color for wave names
- **datacolor** – Color for wave data text
- **nodecolor** – Color for node text
- **gridcolor** – Color of background grid
- **edgecolor** – Color of edge notations (default blue)
- **tickcolor** – Color of tick/tock labels in head/foot
- **grid** – Enable grid lines (default True)

class schemdraw.logic.bitfield.**BitField**(*reg*, ***kwargs*)

Draw a Bit Field compatible with WaveDrom syntax. For reg parameters and examples, see https://github.com/wavedrom/bitfield.

Parameters

- **reg** (dict) – The register dictionary. See below and https://github.com/wavedrom/bitfield.
- **bitheight** – Height of a bit register box in drawing units
- **width** – Full width of the register box in drawing units
- **fontsize** – Size of all text labels
- **lw** – Line width for borders
- **ygap** – Distance between lanes. Omit to auto-space based on label heights

- **vflip** – Flip order of bits
- **hflip** – Flip order of lanes
- **compact** – Remove whitespace between lanes

The reg dictionary may have two keys. 'reg' is a list bitfields, and 'config' tha defines configuration options. Items in the reg list are dictionaries that may include:

- name: Text to display within the bit group
- bits: Number of bits within the group
- **attr: Label to show below the group. May be a string, or integer. If integer,** the binary representation is shown. May also be a list of multiple lines.
- type: 0-9 code to fill the bit group. Or may be any valid color string.

The config list may include:

- lanes: Number of lanes (bit words stacked vertically)
- hflip: Reverse order of lanes
- vflip: Reverse order of bits
- compact: Remove whitespace between lanes
- bits: Total number of bits to include (padded out if not included in the *reg* list)
- label: Dictionary of either 'left' or 'right' and text to display left or right of the lanes.

Schemdraw's implementation has these known differences:

- **'type' parameter, which is used to specify a fill color, can** be the 0-9 code as in WaveDrom, or any valid color string
- hspace defines the full width of the register in pixels, without including any labels
- vspace defines the full width of a register in pixels, without including any labels or padding
- margins are ignored (but can be set by adding the BitField to a schemdraw Drawing)

6.10 Digital Signal Processing

Signal processing elements

class schemdraw.dsp.dsp.**Adc**(**args*, ***kwargs*)

Analog to digital converter

Anchors:

- in
- out
- E (same as in)
- W (same as out)

class schemdraw.dsp.dsp.**Amp**(**args*, ***kwargs*)

Amplifier

Anchors:

- in

- out

class schemdraw.dsp.dsp.**Circle**(**args*, ***kwargs*)

Empty circle element

Anchors:

- N
- S
- E
- W
- NW
- NE
- SW
- SE

class schemdraw.dsp.dsp.**Circulator**(*circle with an arrow in it*)

Anchors:

- N
- S
- E
- W

class schemdraw.dsp.dsp.**Dac**(**args*, ***kwargs*)

Digital to analog converter

Anchors:

- in
- out
- E (same as in)
- W (same as out)

class schemdraw.dsp.dsp.**Demod**(**args*, ***kwargs*)

Demodulator (box with a diode in it)

Anchors:

- N
- S
- E
- W

class schemdraw.dsp.dsp.**Filter**(*response=None*, ***kwargs*)

Parameters

response (Optional[Literal['lp', 'bp', 'hp', 'notch']]) – Filter response ('lp', 'bp', 'hp', or 'notch') for low-pass, band-pass, high-pass, and notch/band-stop filters

Anchors:

- N
- S
- E
- W

class schemdraw.dsp.dsp.**Isolator**(*box with an arrow in it*)

Anchors:

- N
- S
- E
- W

class schemdraw.dsp.dsp.**Mixer**(*N=None*, *E=None*, *S=None*, *W=None*, *font=None*, *fontsize=10*, ***kwargs*)

Parameters

- **N** (Optional[str]) – text in North sector
- **S** (Optional[str]) – text in South sector
- **E** (Optional[str]) – text in East sector
- **W** (Optional[str]) – text in West sector
- **font** (Optional[str]) – Font family/name
- **fontsize** (float) – Point size of label font

Anchors:

- N
- S
- E
- W
- NW
- NE
- SW
- SE

class schemdraw.dsp.dsp.**Oscillator**(**args*, ***kwargs*)

Oscillator in a circle

Anchors:

- N
- S
- E
- W
- NW

- NE
- SW
- SE

class schemdraw.dsp.dsp.**OscillatorBox**(**args*, ***kwargs*)

Oscillator in a square

Anchors:

- N
- S
- E
- W

class schemdraw.dsp.dsp.**Speaker**(**args*, ***kwargs*)

Speaker with only one terminal

class schemdraw.dsp.dsp.**Square**(**args*, ***kwargs*)

Empty square element

Anchors:

- N
- S
- E
- W

class schemdraw.dsp.dsp.**Sum**(**args*, ***kwargs*)

Summation element (+ symbol)

Anchors:

- N
- S
- E
- W
- NW
- NE
- SW
- SE

class schemdraw.dsp.dsp.**SumSigma**(**args*, ***kwargs*)

Summation element (Greek Sigma symbol)

Anchors:

- N
- S
- E
- W

- NW
- NE
- SW
- SE

class schemdraw.dsp.dsp.**VGA**(*tuneup=True*, ***kwargs*)

Variable Gain Amplifier (amplifier symbol with an arrow over it)

Parameters
tuneup (bool) – Set tune above or below the symbol

Anchors:

- input
- out
- tune

6.11 Pictorial Elements

Pictorial Style Elements

class schemdraw.pictorial.pictorial.**Breadboard**(*, *shadow_color=None*, *text_color=None*, ***kwargs*)

Solderless Breadboard, 400-point (30 rows)

Use .color() to set border color, and .fill() to set fill color.

class schemdraw.pictorial.pictorial.**CapacitorCeramic**(*, *radius=None*, *lead_length=None*, ***kwargs*)

Ceramic Disc Capacitor

class schemdraw.pictorial.pictorial.**CapacitorElectrolytic**(*, *lead_length=None*, *cap_color=None*, *stripe_color=None*, ***kwargs*)

Electrolytic capacitor

Note: Use .fill() to change color.

Parameters

- **cap_color** (Optional[str]) – Color of the metal cap on top
- **stripe_color** (Optional[str]) – Color of the polarity stripe

class schemdraw.pictorial.pictorial.**CapacitorMylar**(*, *lead_length=None*, ***kwargs*)

Mylar Capacitor

class schemdraw.pictorial.pictorial.**DIP**(*npins=8*, *wide=False*, ***kwargs*)

Dual-Inline-Package IC

Parameters

- **npins** (int) – Total number of pins
- **wide** (bool) – Use wide (0.6 inch) package instead of narrow (0.3 inch) package

class schemdraw.pictorial.pictorial.**Diode**(*, *stripe_color=None*, ***kwargs*)

Diode in DO-204 package

class schemdraw.pictorial.pictorial.**ElementPictorial**(***kwargs*)

This class gives pictorial 2-term elements silver lead extensions

class schemdraw.pictorial.pictorial.**LED**(**args*, ***kwargs*)

Light Emitting Diode Use .fill() to set the color.

Suggested fill colors for common LEDs:

- red: #dd4433,
- orange: #efa207
- yellow: #f1de0f
- green: #4be317
- blue: #3892bc
- white/clear: #e5e5e5

class schemdraw.pictorial.pictorial.**LEDBlue**(**args*, ***kwargs*)

Blue Light Emitting Diode

class schemdraw.pictorial.pictorial.**LEDGreen**(**args*, ***kwargs*)

Green Light Emitting Diode

class schemdraw.pictorial.pictorial.**LEDOrange**(**args*, ***kwargs*)

Orange Light Emitting Diode

class schemdraw.pictorial.pictorial.**LEDWhite**(**args*, ***kwargs*)

White Light Emitting Diode

class schemdraw.pictorial.pictorial.**LEDYellow**(**args*, ***kwargs*)

Yellow Light Emitting Diode

class schemdraw.pictorial.pictorial.**Resistor**(*value=1000*, *tolerance=None*, ***kwargs*)

Carbon-film 1/4 Watt Resistor

Parameters

- **value** (float) – Resistance value, used to set color bands
- **tolerance** (Optional[float]) – Tolerance value for 4th color band

Note: Color bands will be closest possible value that can be represented with 3 bands

class schemdraw.pictorial.pictorial.**TO92**(**args*, ***kwargs*)

TO92 Transistor Package, with pins bent out to 0.1 inch breadboard spacing

schemdraw.pictorial.pictorial.**parse_size_to_units**(*value*)

Parse a SVG size string (sunh as '2in') to drawing units

Return type
float

schemdraw.pictorial.pictorial.**resistor_colors**(*value*, *tolerance=None*)

Determine Resistor color bands given the value (ohms) and tolerance (in percent)

Return type
tuple[str, str, str, Optional[str]]

class schemdraw.pictorial.fritz.**FritzingPart**(*fname*, *partname=None*, *partidx=None*, *scale=1.0*)

Load a Fritzing Part File as a Schemdraw Element

Anchors will be extracted from the Part definition file. Note some anchors are not valid Python identifiers and therefore must be accessed through the Element.absanchors dictionary rather than an attribute of the element instance.

Parameters

- **fname** (str) – Filename of fritzing .fzpz archive
- **partname** (Optional[str]) – Name of part within the file. Use *listparts* to show all names. If not provided, first part is drawn.
- **partidx** (Optional[int]) – Index of part within the file. If not provided, first part is drawn. Overrides *partname*.
- **scale** (float) – Scale factor

6.12 Flowcharting

Flowcharting element definitions

class schemdraw.flow.flow.**Box**(***kwargs*)

Flowchart Process Box. Default box has minimum size (3, 2) but will expand to fit the label text. Size may be manually fixed using w and h arguments.

Parameters

- **w** – Width of box
- **h** – Height of box

Anchors:

- 16 compass points (N, S, E, W, NE, NNE, etc.)

schemdraw.flow.flow.**Circle**

alias of *Connect*

class schemdraw.flow.flow.**Connect**(***kwargs*)

Flowchart connector/circle

Parameters
r – Radius of circle

Anchors:

- 16 compass points (N, S, E, W, NE, NNE, etc.)

class schemdraw.flow.flow.**Data**(*, *slant=0.5*, ***kwargs*)

Flowchart data or input/output box (parallelogram)

Parameters

- **w** – Width of box
- **h** – Height of box
- **s** – slant of sides

Anchors:

- 16 compass points (N, S, E, W, NE, NNE, etc.)

class schemdraw.flow.flow.**Decision**(*N=None*, *E=None*, *S=None*, *W=None*, *font=None*, *fontsize=14*, ***kwargs*)

Flowchart decision (diamond)

Parameters

- **w** – Width of box
- **h** – Height of box
- **N** (Optional[str]) – text for North decision branch
- **S** (Optional[str]) – text for South decision branch
- **E** (Optional[str]) – text for East decision branch
- **W** (Optional[str]) – text for West decision branch
- **font** (Optional[str]) – Font family/name
- **fontsize** (float) – Point size of label font

Anchors:

- 16 compass points (N, S, E, W, NE, NNE, etc.)

class schemdraw.flow.flow.**Ellipse**(***kwargs*)

Flowchart ellipse

Parameters

- **w** – Width of ellipse
- **h** – Height of ellipse

Anchors:

- 16 compass points (N, S, E, W, NE, NNE, etc.)

schemdraw.flow.flow.**Process**

alias of *Box*

class schemdraw.flow.flow.**RoundBox**(***, *cornerradius=None*, ***kwargs*)

Alternate Process box with rounded corners

Parameters

- **w** – Width of box
- **h** – Height of box
- **cornerradius** (Optional[float]) – Radius of round corners [default: 0.3]

Anchors:

- 16 compass points (N, S, E, W, NE, NNE, etc.)

schemdraw.flow.flow.**RoundProcess**

alias of *RoundBox*

schemdraw.flow.flow.**Start**

alias of *Terminal*

schemdraw.flow.flow.**State**

alias of *Connect*

class schemdraw.flow.flow.**StateEnd**(***kwargs*)

End/Accept State (double circle)

Parameters

- **r** – radius
- **dr** – distance between circles

Anchors:

- 16 compass points (N, S, E, W, NE, NNE, etc.)

class schemdraw.flow.flow.**Subroutine**(*, *s=None*, ***kwargs*)

Flowchart subroutine/predefined process. Box with extra vertical lines near sides.

Parameters

- **w** – Width of box
- **h** – Height of box
- **s** (Optional[float]) – spacing of side lines [default: 0.3]

Anchors:

- 16 compass points (N, S, E, W, NE, NNE, etc.)

class schemdraw.flow.flow.**Terminal**(**args*, ***kwargs*)

Flowchart start/end terminal

Parameters

- **w** – Width of box
- **h** – Height of box

Anchors:

- 16 compass points (N, S, E, W, NE, NNE, etc.)

schemdraw.flow.flow.**labelsize**(*label*, *pad*)

Get unit size of label and padding

CHAPTER

SEVEN

CHANGE LOG

v0.23 - 2026-05-29

- Added BitField register diagrams compatible with WaveDROM syntax
- Fixed label alignment on flipped and/or reversed IC elements
- SVG backend implements rounding of coordinates to n (default 3) significant figures
- Prevent matplotlib window when *show* is False
- Renamed AnalogNFet and AnalogPFet arrow parameter to *source_arrow* to prevent naming clash
- Added ground anchors to OutletF.
- **Enhancements to TimingDiagram:**
  - Implement variable-height signals using *level* parameter
  - Added *nodealign* parameter to align nodes to clock instead of risetime
  - Added *fontsize* parameter to each signal
  - Added differential clock *Q* waveform type
  - Added half-period bit *b* waveform type
  - Define shading of periods
  - Allow signal with no wave, for drawing data text only
  - Improve alignment of edge text and arrows

v0.22 - 2025-11-30

Enhancements:

- Added vacuum tube elements
- Added box-style meters
- Added parameter to remove contact circles from switches
- Added Voltage label as an arc over an element
- Added *shift* method to offset a 2-terminal element toward one end or the other
- Updated *tox* and *toy* methods to work with diagonal placement
- Updated *Transformer* to allow multi-phase connections
- Option to draw core lines on Inductor and Inductor2, and defined NE, NW, SE, SW anchors for placing mutual inductance dots.
- Elements can define label position hints for fine-tuned placement of labels on anchor positions

- Added *Drawing.hold()* to create context manager that saves and restores the drawing state

New Elements:

- VacuumTube
- Triode
- Tetrode
- Pentode
- DualVacuumTube
- TubeDiode
- BatteryDouble
- MeterArrow
- VoltageLabelArc
- MeterAnalog
- MeterDigital
- Oscilloscope

v0.21 - 2025-07-27

Enhancements:

- Improved documentation to make elements searchable
- Fixed SVG namespaces when importing Fritzing files
- Renamed DB9 connector to DE9, with DB9 as alias (see https://news.sparkfun.com/14298)
- Absolute anchor positions (relative to drawing coordinates) may be accessed using index notation on the element (e.g. *mybjt['base']*)
- Added gradient fill method
- Support hyperlink labels in SVG output (thanks to masa300V)

New Elements:

- Hemt
- DiodeTVS
- NpnSchottky
- PnpSchottky
- NpnPhoto
- PnpPhoto
- IgbtN
- IgbtP
- DA15
- DC37
- DD50

v0.20 - 2025-03-08

Enhancements:

- Added *transparent* and *dpi* parameters to *Drawing* constructor, to be used when saving to a file from the context manager
- Validate colors and linestyles when set instead of waiting for matplotlib to draw, or SVG to silently fail
- Implemented *head* and *foot* parameters of wavedrom TimingDiagrams
- Added *box* parameter to Josephson element

Fixes:

- Gate anchor position on Pmos2
- Updated *Arrow* docstring with correct *arrowwidth* and *arrowlength* parameters
- Fixed Arc arrowheads post-transformation
- Fixed CurrentLabel arrow offset
- Fix user-supplied label rotation angles when going upside-down
- Clear the figure when calling *draw* to prevent double-drawing
- Fix arrow direction in timing diagrams
- Adjust label position along Arcs

v0.19 - 2024-04-27

Enhancements:

- Added *scalex* and *scaley* for asymmetric zooming/scaling of Elements
- Added ElementImage for placing PNG and SVG images as an Element
- Added pictorial elements
- Load Fritzing Part files (breadboard view)
- Added *.pin* and *.side* methods to Ic Element
- Added *mathfont* parameter to drawing config method

Fixes:

- Fixed *tox* and *toy* methods when the Elemnet defines a theta
- Fixed vertical offset of labels below an Element
- Fixed reversing of CurrentLabel.
- Update Ic Element to respect *pinspacing* when *size* is also provided
- SVG backend accepts color tuples (r, g, b) in the range 0-1, mirroring Matplotlib
- Tighten bounding box around text elements (when using ziamath)

v0.18 - 2023-12-29

Enhancements:

- Automatically increase size of flowchart boxes to fit the text
- Added Container element to automatically draw a box around a group of elements
- Internally track drawing state so that *d.add* or *d* += is no longer necessary when adding elements inside a context manager

- Default element attributes may be set using *Element.defaults* dictionary. For example, to fill all Diodes, set *elm.Diode.defaults['fill'] = True*.

New Elements:

- Lamp2
- Terminal

Other changes:

- BREAKING: Subclasses of ElementCompound must add Segments inside a *setup* method instead of in *__init__*. The *inner_element* and *outer_element* parameters of *TwoPort* now accept classes instead of instances of those classes.

v0.17 - 2023-06-03

New Elements:

- Tristate inverter (credit: Jan Genoe)
- NMos and PMos elements (credit: dtmaidenmueller)
- AnalogNFet, AnalogPFet, AnalogBiasedFet
- DataBusLine
- CurrentMirror, VoltageMirror
- Nullator, Norator, VMCMPair
- **Compound twoport elements (ElementTwoport base class):**
  - TwoPort
  - VoltageTransactor
  - TransimpedanceTransactor
  - CurrentTransactor
  - TransadmittanceTransactor
  - Nullor

Enhancements:

- Added arguments for length/width of CurrentLabel arrows (credit: Christian Seigel)
- Added config option for setting whitespace margins

Bug fixes:

- Fixed regression bug in logicparse where labels were not drawn on inputs/outputs
- Fix timing diagrams when async times are longer than the wave
- Fixed default style hierarchy on segments
- Fixed CurrentLabel placement with transistor elements, now follows biasing current
- Fixed CurrentLabel positioning for elements with no center anchor

Other changes:

- Deprecated positional direction parameter to Element class.
- Add nonglobal rotation mode attribute to SegmentText

v0.16 - 25-Mar-2023

- Added *canvas* parameter to Drawing and draw method, and deprecated *backend* parameter.
- Removed *elements* argument from Drawing. Use Drawing.add_elements.
- Added *Drawing.set_anchor* to define anchor points, useful for ElementDrawing instances.
- Added shunt resistor symbol Rshunt
- Fixed lead length of XOR gates to align with OR gates
- Fixed greater and less than symbols in SVG backend
- Fixed some anchor positions on flowchart symbols
- Allow font parameter to be the path of a TTF file
- Removed old deprecations
- Replaced setup.py with setup.cfg

v0.15 - 20-Jun-2022

- Added DSP elements Circulator, Isolator, VGA
- Added ZLabel element for right-angle impedance arrow labels
- Changed DSP anchor names from 'in' to 'input' to avoid conflict with 'in' keyword
- Fixed styles with nested ElementDrawing elements
- Fixed zorder of filled elements in Matplotlib backend
- Added mathfont parameter to labels for specifying different font on math labels
- Added padx and pady parameters to Encircle elements
- Moved SVG backend configuration to svgconfig object and deprecated 'settextmode'.
- Added 'visible' parameter to Segment objects

v0.14 - 09-Jan-2022

- Added context manager to Drawing class.
- Added Wire element for quick 90-degree connections
- Added Encircle, EncircleBox, and Annotate elements
- Added Wheatstone, Rectifier, SparkGap elements
- Added "2T" version of transistor elements for placement as as two-terminal elements
- *tox* and *toy* methods automatically change drawing direction, removing need to specify *right()* and *tox()*, for example.
- Added *istart* and *iend* anchors to 2-Terminal elements for defining inner start and end points before lead extensions
- Added *dot* and *idot* methods to two-terminal elements
- Added '-o' and '-|' arrow types to draw dot or endcap instead of arrow at the end of lines
- Added *leads* parameter to OpAmp for adding lead extensions
- Added *lead* parameter to Grounds, Vss, and Vdd
- Added optional *dx* and *dy* parameters to *to* and *at* methods for quick fine-tuning of placement
- Added optional *length* parameter to *up*, *down*, *left*, and *right* on two-terminal elements
- Improved placement of CurrentLabel arrows

- Fix default label position on Vss element
- Fix positioning of switch contact bubbles
- Fix text rotation in svg backend and path mode
- The *scale* method now maintains the length of two-terminal elements

v0.13 - 19-Dec-2021

- Added Digital Timing Diagram elements
- Added Table and Kmap elements
- Added Arc2, Arc3, ArcN, ArcZ, ArcLoop elements, useful for state machine diagrams
- Added drop method to Element class to specify where to leave the drawing position
- Added move_from method to Drawing class to move relative to another element anchor
- Added more anchors to all flowchart elements
- Improved layout of flowchart elements. **May affect layout of some existing flowchart diagrams.**
- Added SegmentBezier for creating elements with curves
- Deprecated SegmentArrow in favor of Segment with arrow parameter

v0.12 - 05-Nov-2021

- Fixed Arrow and LineDot element placement when placed with anchor
- Fixed copy/pickle of Element class
- Fixed importing * from schemdraw.elements

v0.11 - 10-Jul-2021

- Fixed placing elements by anchor when anchors were defined using a tuple rather than Point
- SVG backend adds option for SVG1.x format for better compatibility with SVG renderers
- Restore compabitliiby with Python 3.7 via conditional import of typing_extensions.

v0.10 - 30-Apr-2021

- Added options to place labels inside Mixer elements
- Fixed arrowhead overshoot
- Fixed get_imagedata function
- Update pip install to include optional dependencies
- Added ziamath optional dependency for rendering math in SVG backend
- Added LoopArrow as superclass of LoopCurrent, for placing a loop anywhere

v0.9.1 - 30-Jan-2021

- Fixed missing module in setup.py.

v0.9 - 30-Jan-2021

- Added optional SVG backend for drawing directly to an SVG image
- Implemented method-chaining "fluent" interface for building elements
- Added *elements.style* method for setting U.S. or European/IEC resistor style
- Added parameter for drawing schematic on existing matplotlib axis

- Added string parser for creating logic diagrams from expressions like "A or B"
- Fixed zooming of arc segments
- Added type annotations
- Added *Drawing.move* method for moving cursor by dx and dy.
- Drawing class implements += operator, so elements can be added by *Drawing += Element()*
- Removed dependency on Numpy
- Added Drawing.interactive` to allow element-by-element drawing with Matplotlib's plt.ion().
- Now requires Python 3.8+
- **New Elements:**
  - CPE (Constant Phase Element)
  - Varactor
  - FuseIEEE
  - FuseIEC
  - SwitchRotary
  - SwitchReed
  - Jack
  - Plug
  - Ic555
  - IcDIP
  - SevenSegment
  - Outlet Elements
- **Deprecations:**
  - *Element.add_label* is deprecated (use *Element.label*)
  - *Drawing.loopI* is deprecated (add a *LoopCurrent* element)
  - *Drawing.labelI* is deprecated (add a *CurrentLabel* element)
  - *Drawing.labelI_inline* is deprecated (add a *CurrentLabelInline* element)

v0.8 - 15-Aug-2020

- Changed Header anchors to *pinN* for consistency with Ic.
- Improved label placement with respect to anchor positions.
- Prevent duplicate figures from showing in Jupyter Element representation
- Improvements for headless server operation to prevent popup window
- Added some undocumented features to documentation
- Added *Drawing.get_imagedata* function for returning raw image bytes
- Fixed pip installation issue with module capitalization. Must import lowercase *schemdraw*.

v0.7.1 - 26-Jun-2020

- Bug fix: restore usage outside of Jupyter, so that Matplotlib window is shown when calling Drawing.draw().

v0.7 - 21-Jun-2020

- Dropped support for Python 2. Now requires 3.7+.
- Elements are now subclasses of Element. Previous (dict) element names are translated into new class names. Any user-defined elements will need to be converted to classes. The *group_elements* function is replaced with *ElementDrawing* class.
- Allow *fontsize* or *size* keyword arguments interchangeably in Drawing and add_label
- Updated flow.Decision to use keyword arguments for labeling decision branches
- The Ic element label offset parameter changed from *lblofst* to *lofst* to avoid conflict with the main element label.
- Direct access to Drawing.fig and Drawing.ax are no longer available. Instead, Drawing.draw() returns a schemdraw.Figure instance with fig and ax attributes.
- Implemented Jupyter representation functions for both Drawing and Element classes.
- **New Elements:**
  - Coax
  - Triax
  - SwitchDpst
  - SwitchDpdt
  - Relay
  - Optocoupler
  - Arrow
  - LineDot
  - Breaker
  - OrthoLines
  - RightLines
  - BusConnect
  - BusLine
  - Tag
  - Photoresistor
  - PhotoresistorBox
  - Thermistor
  - DiodeShockley
  - PotBox
  - RBoxVar
  - Solar
  - Neon
  - SourceSquare
  - AntennaLoop
  - AntennaLoop2

  - AudioJack
  - Tgate
  - Schmitt
  - SchmittNot
  - SchmittAnd
  - SchmittNand

v0.6.0, 11-Feb-2020

- Refactored internals to allow more control over individual components of drawing. Should have no effect unless the user is accessing internal attributes of the Element object. This also adds the *segments* list to the Element object, which allows finer control over individual bits of the drawing.
- Updated add_label so that "top" labels should always appear on top, regardless of flip/reverse
- Swapped the direction of current sources, so that a current source with direction "up" has the arrow pointing up.
- Added "zorder" parameter in the element definition dictionary and *add* method
- Added *elements.ic* and *elements.multiplexer* functions as replacements to *blackbox* and *mux*. These include more functionality such as adjusting indiviudal pin rotation, color, and inverter bubbles.
- Labels can be placed relative to an anchor position using the *add_label* method. This could be useful, for example, in labeling pin numbers on a logic gate or opamp.
- Added new anchors to OPAMPs for power supply and offset nulls.
- **New Elements:**
  - MIC
  - MOTOR
- **Documentation:**
  - Upgraded documentation to Sphinx and moved to readthedocs.org at https://schemdraw.readthedocs.io/en/latest/.
  - Changed preferred import to *import SchemDraw.elements as elm*. Apparently some people still use import * with pylab; this suggestion will help avoid conflicts.

v0.5.0, 21-Jul-2019

- Added flowcharting symbol methods to SchemDraw.flow module
- Added signal processing symbols to SchemDraw.dsp module
- Implemented fill parameter on Drawing.add to fill shapes and closed paths with a solid color
- **New elements:**
  - Fuse
  - CapacitorVar,
  - DiodeTunnel
  - Jfet
  - Diac
  - Triac
  - SCR

v0.4.0, 03-Nov-2018

- Fixed drawing of NOT and related gates to property extend the path
- Fixed arrow translation when grouping elements
- Fixed sidelabels and plabels of blackbox when empty
- Fixed arc drawing due to change in Matplotlib 2.2 on asymmetric partial arcs

v0.3.0, 03-Jul-2017

- Added function for drawing multiplexers/demultiplexers
- Updates to labelI() method to allow reversing arrow and changing length
- Add CSS to documentation
- **New elements:**
  - PHOTODIODE
  - NFET4
  - PFET4
  - VSS
  - VDD

v0.2.2, 06-Mar-2016

- Documentation updates
- **New elements:**
  - Transformer
  - Josephson Junction (JJ)

v0.2.1, 03-May-2015

- Fixed anchor names when element overwrites base anchor, such as BJT_PNP.
- Added showplot keyword to draw() for non-interactive mode.
- Added 2-collector BJT.
- Documentation: added gallery of schematics.

v0.2.0, 29-Apr-2015

- Added default line width argument to drawing() class. Default width is now 1.5.
- Converted documentation to use all vector-based images
- Added XKCD-mode example
- **New elements:**
  - BATTERY
  - BAT_CELL
  - SPEAKER
  - BUTTON
  - BUTTON_NC
  - XTAL

  - MEMRISTOR,
  - SCHOTTKY
  - ZENER
  - LED2

v0.1.4, 30-Sep-2014

- Add function to group several elements into one
- Add blackbox() function to generate box elements with arbitrary inputs
- Allow element definition to specify label alignment
- Added linestyle to element kwargs and definition
- **New elements:**
  - LED
  - OPAMP_NOSIGN
  - GAP_LABEL
  - ELLIPSIS

v0.1.3, 21-Sep-2014

- Added logic gate elements
- Added transparent and dpi options to save() function
- Fixed issues with zooming and rotating elements with arcs
- LaTeX typesetting uses sans-serif, regular fonts for consistency

v0.1.0, 25-Aug-2014

- Initial Release

CHAPTER

EIGHT

DEVELOPMENT

Code contributions to schemdraw are welcome, especially with:

- New circuit elements
- Building non-electrical component libraries (for example, process flow diagrams would be a great addition)
- Adding examples to the documentation gallery

Report bugs and feature requests on the Issue Tracker. Please include code for an example schematic drawing affected by the bug.

For simple bug fixes, if you're comfortable working a fix and submitting a pull request (PR), please fork the Source Code and issue the (PR). Simple fixes are typically those affecting a few lines in one source file.

For more involved bug fixes, or adding new features and new elements, please open an issue in the Issue Tracker. before submitting a PR. This allows discussion on the best possible solution.

Make sure to include any new elements in an appropriate location in the test Jupyter notebooks and in the documentation.

8.1 Guidelines

- All contributions will be included under schemdraw's MIT license.
- Schemdraw generally follows PEP 8 code conventions. Please match code style in PRs to the schemdraw style. PRs with superfluous formatting changes, especially where it is difficult to separate the functional from formatting differences, will not be accepted.
- Code, documentation, or issues generated by an LLM, chatbot, or other code generation bot will not be accepted at this time.

8.2 Citing Schemdraw

Are you using schemdraw in digital or printed publication and want to show it off? Great! Share your work by adding an item in the github Discussion section. Once enough examples are collected, links may be added in the documentation.

You do not need to cite schemdraw in your publication unless the publication directly discusses use of schemdraw as a schematic drawing tool. If you'd like to cite, one way is to refer to the documentation PDF. Using APA style:

> Delker, Collin (2026). Schemdraw Documentation. readthedocs. Retrieved from https://schemdraw.readthedocs.io/_/downloads/en/stable/pdf/

Or a bibtex entry:

```
@misc{delker2026schemdraw, title={Schemdraw Documentation}, author={Delker, Collin},
year={2026}, publisher={readthedocs}, url={https://schemdraw.readthedocs.io/_/downloads/en/stable/
pdf/} }
```

Want to support Schemdraw development? Need more circuit examples? Pick up the Schemdraw Examples Pack on buymeacoffee.com:

PYTHON MODULE INDEX

S

INDEX

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z