skidl Documentation

Release 0.0.7

XESS Corp.

January 11, 2017

Contents

CHAPTER 1

skidl

SKiDL is a module that allows you to compactly describe the interconnection of electronic circuits and components using Python. The resulting Python program performs electrical rules checking for common mistakes and outputs a netlist that serves as input to a PCB layout tool.

- Free software: MIT license
- Documentation: http://xesscorp.github.io/skidl

1.1 Features

- Has a powerful, flexible syntax (because it *is* Python).
- Permits compact descriptions of electronic circuits (think about *not* tracing signals through a multi-page schematic).
- Allows textual descriptions of electronic circuits (think about using `diff` and git for circuits).
- Performs electrical rules checking (ERC) for common mistakes (e.g., unconnected device I/O pins).
- Supports linear / hierarchical / mixed descriptions of electronic designs.
- Fosters design reuse (think about using PyPi and Github to distribute electronic designs).
- Makes possible the creation of *smart circuit modules* whose behavior / structure are changed parametrically (think about filters whose component values are automatically adjusted based on your desired cutoff frequency).
- Can work with any ECAD tool (only two methods are needed: one for reading the part libraries and another for outputing the correct netlist format).
- Takes advantage of all the benefits of the Python ecosystem (because it *is* Python).

As a very simple example, the SKiDL program below describes a circuit that takes an input voltage, divides it by three, and outputs it:

```
from skidl import *

gnd = Net('GND')  # Ground reference.
vin = Net('VI')   # Input voltage to the divider.
vout = Net('VO')  # Output voltage from the divider.
r1, r2 = 2 * Part('device', 'R', TEMPLATE)  # Create two resistors.
r1.value, r1.footprint = '1K',  'Resistors_SMD:R_0805'  # Set resistor values
r2.value, r2.footprint = '500', 'Resistors_SMD:R_0805'  # and footprints.
r1[1] += vin      # Connect the input to the first resistor.
r2[2] += gnd      # Connect the second resistor to ground.
```

```
vout += r1[2], r2[1]  # Output comes from the connection of the two resistors.

generate_netlist()
```

And this is the output that can be fed to a program like KiCad's PCBNEW to create the physical PCB:

```
(export (version D)
  (design
    (source "C:/Users/DEVB/PycharmProjects/test1\test.py")
    (date "08/12/2016 11:13 AM")
    (tool "SKiDL (0.0.1)"))
  (components
    (comp (ref R1)
      (value 1K)
      (footprint Resistors_SMD:R_0805))
    (comp (ref R2)
      (value 500)
      (footprint Resistors_SMD:R_0805)))
  (nets
    (net (code 0) (name "VI")
      (node (ref R1) (pin 1)))
    (net (code 1) (name "GND")
      (node (ref R2) (pin 2)))
    (net (code 2) (name "VO")
      (node (ref R1) (pin 2))
      (node (ref R2) (pin 1))))
)
```

CHAPTER 2

Caveats

Before working with SKiDL, please realize that it is still under development and features are missing and/or likely to change. The most notable missing feature is the lack of *back annotation* where changes made to a circuit during PCB layout are reflected back into the schematic (or, in our case, the SKiDL program).

Also, while SKiDL-generated netlists have been brought into KiCad's PCBNEW layout editor and manipulated, no physical PCBs have yet been fabricated.

CHAPTER 3

Installation

SKiDL is pure Python so it's easy to install:

```
$ pip install skidl
```

or:

```
$ easy_install skidl
```

CHAPTER 4

Usage

4.1 Scratching the Surface

This is the minimum that you need to know to design electronic circuitry using SKiDL:

- How to get access to SKiDL.
- How to find and instantiate a component (or *part*).
- How to connect *pins* of the parts to each other using *nets*.
- How to run an ERC on the circuit.
- How to generate a *netlist* for the circuit that serves as input to a PCB layout tool.

I'll demonstrate these steps using SKiDL in an interactive Python session, but normally the statements that are shown would be entered into a file and executed as a Python script.

4.1.1 Accessing SKiDL

To use skidl in a project, just place the following at the top of your file:

```
import skidl
```

But for this tutorial, I'll just import everything:

```
from skidl import *
```

4.1.2 Finding Parts

SKiDL provides a convenience function for searching for parts called (naturally) `search`. For example, if you needed an operational amplifier, then the following command would pull up some likely candidates:

```
>>> search('opamp')
linear.lib: LT1492
linear.lib: MCP601R
linear.lib: LT1493
linear.lib: MCP603
linear.lib: LM4250
...
linear.lib: LM386
linear.lib: MCP603SN
```

```
linear.lib: INA128
linear.lib: LTC6082
linear.lib: MCP601SN
```

`search` accepts a regular expression and scans for it *anywhere* within the name, description and keywords of all the parts in the library path. So the following search pulls up several candidates:

```
>>> search('lm35')
dc-dc.lib: LM3578
linear.lib: LM358
regul.lib: LM350T
sensors.lib: LM35-NEB
sensors.lib: LM35-D
sensors.lib: LM35-LP
```

If you want to restrict the search to a specific part, then use a regular expression like the following:

```
>>> search('^lm358$')
linear.lib: LM358
```

Once you have the part name and library, you can see the part's pin numbers, names and their functions using the `show` function:

```
>>> show('linear', 'LM358')
LM358:
        Pin 4/V-: POWER-IN
        Pin 8/V+: POWER-IN
        Pin 1/~: OUTPUT
        Pin 2/-: INPUT
        Pin 3/+: INPUT
        Pin 5/+: INPUT
        Pin 6/-: INPUT
        Pin 7/~: OUTPUT
```

`show` looks for exact matches of the part name in a library, so the following command raises an error:

```
>>> show('linear', 'lm35')
ERROR: Unable to find part lm35 in library linear.
```

4.1.3 Instantiating Parts

The part library and name are used to instantiate a part as follows:

```
>>> resistor = Part('device','R')
```

You can customize the resistor by setting its attributes:

```
>>> resistor.value = '1K'
>>> resistor.value
'1K'
```

You can also combine the setting of attributes with the creation of the part:

```
>>> resistor = Part('device', 'R', value='1K')
>>> resistor.value
'1K'
```

You can use any valid Python name for a part attribute, but `ref`, `value`, and `footprint` are necessary in order to generate the final netlist for your circuit. And the attribute can hold any type of Python object, but simple strings are probably the most useful.

The `ref` attribute holds the *reference* for the part. It's set automatically when you create the part:

```
>>> resistor.ref
'R1'
```

Since this was the first resistor we created, it has the honor of being named `R1`. But you can easily change it:

```
>>> resistor.ref = 'R5'
>>> resistor.ref
'R5'
```

Now what happens if we create another resistor?:

```
>>> another_res = Part('device','R')
>>> another_res.ref
'R1'
```

Since the `R1` reference was now available, the new resistor got it. What if we tried renaming the first resistor back to `R1`:

```
>>> resistor.ref = 'R1'
>>> resistor.ref
'R1_1'
```

Since the `R1` reference was already taken, SKiDL tried to give us something close to what we wanted. SKiDL won't let different parts have the same reference because that would confuse the hell out of everybody.

4.1.4 Connecting Pins

Parts are great and all, but not very useful if they aren't connected to anything. The connections between parts are called *nets* (think of them as wires) and every net has one or more part *pins* on it. SKiDL makes it easy to create nets and connect pins to them. To demonstrate, let's build the voltage divider circuit shown in the introduction.

First, start by creating two resistors (note that I've also added the `footprint` attribute that describes the physical package for the resistors):

```
>>> rup = Part('device', 'R', value='1K', footprint='Resistors_SMD:R_0805')
>>> rlow = Part('device', 'R', value='500', footprint='Resistors_SMD:R_0805')
>>> rup.ref, rlow.ref
('R1', 'R2')
>>> rup.value, rlow.value
('1K', '500')
```

To bring the voltage that will be divided into the circuit, let's create a net:

```
>>> v_in = Net('VIN')
>>> v_in.name
'VIN'
```

Now attach the net to one of the pins of the `rup` resistor (resistors are bidirectional which means it doesn't matter which pin, so pick pin 1):

```
>>> rup[1] += v_in
```

You can verify that the net is attached to pin 1 of the resistor like this:

```
>>> rup[1].net
VIN: Pin 1/~: PASSIVE
```

Next, create a ground reference net and attach it to `rlow`:

```
>>> gnd = Net('GND')
>>> rlow[1] += gnd
>>> rlow[1].net
GND: Pin 1/~: PASSIVE
```

Finally, the divided voltage has to come out of the circuit on a net. This can be done in several ways. The first way is to define the output net and then attach the unconnected pins of both resistors to it:

```
>>> v_out = Net('VO')
>>> v_out += rup[2], rlow[2]
>>> rup[2].net, rlow[2].net
(VO: Pin 2/~: PASSIVE, Pin 2/~: PASSIVE, VO: Pin 2/~: PASSIVE, Pin 2/~: PASSIVE)
```

An alternate method is to connect the resistors and then attach their junction to the output net:

```
>>> rup[2] += rlow[2]
>>> v_out = Net('VO')
>>> v_out += rlow[2]
>>> rup[2].net, rlow[2].net
(VO: Pin 2/~: PASSIVE, Pin 2/~: PASSIVE, VO: Pin 2/~: PASSIVE, Pin 2/~: PASSIVE)
```

Either way works! Sometimes pin-to-pin connections are easier when you're just wiring two devices together, while the pin-to-net connection method excels when three or more pins have a common connection.

4.1.5 Checking for Errors

Once the parts are wired together, you can do simple electrical rules checking like this:

```
>>> ERC()

2 warnings found during ERC.
0 errors found during ERC.
```

Since this is an interactive session, the ERC warnings and errors are stored in the file `skidl.erc`. (Normally, your SKiDL circuit description is stored as a Python script such as `my_circuit.py` and the `ERC()` function will dump its messages to `my_circuit.erc`.) The ERC messages are:

```
WARNING: Only one pin (PASSIVE pin 1/~ of R/R1) attached to net VIN.
WARNING: Only one pin (PASSIVE pin 1/~ of R/R2) attached to net GND.
```

These messages are generated because the `VIN` and `GND` nets each have only a single pin on them and this usually indicates a problem. But it's OK for this simple example, so the ERC can be turned off for these two nets to prevent the spurious messages:

```
>>> v_in.do_erc = False
>>> gnd.do_erc = False
>>> ERC()

No ERC errors or warnings found.
```

4.1.6 Generating a Netlist

The end goal of using SKiDL is to generate a netlist that can be used with a layout tool to generate a PCB. The netlist is output as follows:

```
>>> generate_netlist()
```

Like the ERC output, the netlist shown below is stored in the file `skidl.net`. But if your SKiDL circuit description is in the `my_circuit.py` file, then the netlist will be stored in `my_circuit.net`.

```
(export (version D)
  (design
    (source "C:\xesscorp\KiCad\tools\skidl\skidl\skidl.py")
    (date "08/12/2016 10:05 PM")
    (tool "SKiDL (0.0.1)"))
  (components
    (comp (ref R1)
      (value 1K)
      (footprint Resistors_SMD:R_0805))
    (comp (ref R2)
      (value 500)
      (footprint Resistors_SMD:R_0805)))
  (nets
    (net (code 0) (name "VIN")
      (node (ref R1) (pin 1)))
    (net (code 1) (name "GND")
      (node (ref R2) (pin 1)))
    (net (code 2) (name "VO")
      (node (ref R1) (pin 2))
      (node (ref R2) (pin 2))))
)
```

You can also generate the netlist in XML format:

```
>>> generate_xml()
```

This is useful in a KiCad environment where the XML file is used as the input to BOM-generation tools.

4.2 Going Deeper

The previous section showed the bare minimum you need to know to design circuits with SKiDL, but doing a complicated circuit that way would suck donkeys. This section will talk about some more advanced features.

4.2.1 Basic SKiDL Objects: Parts, Pins, Nets, and Buses

SKiDL uses four types of objects to represent a circuit: `Part`, `Pin`, `Net`, and `Bus`.

The `Part` object represents an electronic component, which SKiDL thinks of as simple bags of `Pin` objects with a few other attributes attached (like the part number, name, reference, value, footprint, etc.).

The `Pin` object represents a terminal that brings an electronic signal into and out of the part. Each `Pin` object has two important attributes:

- `part` which stores the reference to the `Part` object to which the pin belongs.
- `net` which stores the the reference to the `Net` object that the pin is connected to, or `None` if the pin is unconnected.

A `Net` object is kind of like a `Part`: it's a simple bag of pins. The difference is, unlike a part, pins can be added to a net. This happens when a pin on some part is connected to the net or when the net is merged with another net.

Finally, a `Bus` is just a list of `Net` objects. A bus of a certain width can be created from a number of existing nets, newly-created nets, or both.

4.2.2 Creating SKiDL Objects

Here's the most common way to create a part in your circuit:

```
my_part = Part('some_library', 'some_part_name')
```

When this is processed, the current directory will be checked for a file called `some_library.lib` which will be opened and scanned for a part with the name `some_part_name`. If the file is not found or it doesn't contain the requested part, then the process will be repeated using KiCad's default library directory. (You can change SKiDL's library search by changing the list of directories stored in the *skidl.lib_search_paths_kicad* list.)

You're not restricted to using only the current directory or the KiCad default directory to search for parts. You can also search any file for a part by using a full file name:

```
my_part = Part('C:/my_libs/my_great_parts.lib', 'my_super_regulator')
```

You're also not restricted to getting an exact match on the part name: you can use a *regular expression* instead. For example, this will find a part with "358" anywhere in a part name or alias:

```
my_part = Part('some_library', '.*358.*')
```

If the regular expression matches more than one part, then you'll only get the first match and a warning that multiple parts were found.

Once you have a part, you can set its attributes like you could for any Python object. As was shown previously, the `ref` attribute will already be set but you can override it:

```
my_part.ref = 'U5'
```

The `value` and `footprint` attributes are also required for generating a netlist. But you can also add any other attribute:

```
my_part.manf = 'Atmel'
my_part.setattr('manf#', 'ATTINY4-TSHR'
```

It's also possible to set the attributes during the part creation:

```
my_part = Part('some_lib', 'some_part', ref='U5', footprint='SMD:SOT23_6', manf='Atmel')
```

Creating nets is also simple:

```
my_net = Net()               # An unnamed net.
my_other_net = Net('Fred')   # A named net.
```

As with parts, SKiDL will alter the name you assign to a net if it collides with another net having the same name.

You can create a bus of a certain width like this:

```
my_bus = Bus('bus_name', 8)  # Create a byte-wide bus.
```

(All buses must be named, but SKiDL will look for and correct colliding bus names.)

You can also create a bus from existing nets, or buses, or the pins of parts:

```
my_part = Part('linear', 'LM358')
a_net = Net()
b_net = Net()
```

```
bus_nets = Bus('net_bus', a_net, b_net)             # A 2-bit bus.
bus_pins = Bus('pin_bus', my_part[1], my_part[3])   # A 2-bit bus.
bus_buses = Bus('bus_bus', my_bus)                  # An 8-bit bus.
```

Finally, you can mix-and-match any combination of widths, nets, buses or part pins:

```
bus_mixed = Bus('mongrel', 8, a_net, my_bus, my_part[2])  # 8+1+8+1 = 18-bit bus.
```

The final object you can create is a `Pin`. You'll probably never do this (except in interactive sessions), and it's probably a mistake if you ever do do it, but here's how to do it:

```
>>> p = Pin(num=1, name='my_pin', func=Pin.TRISTATE)
>>> p
Pin 1/my_pin: TRISTATE
```

4.2.3 Copying SKiDL Objects

Instead of creating a SKiDL object from scratch, sometimes it's easier to just copy an existing object. Here are some examples of creating a resistor and then making some copies of it:

```
>>> r1 = Part('device', 'R', value=500)
>>> r2 = r1.copy()                          # Make a single copy of the resistor.
>>> r3 = r1.copy(value='1K')                # Make a single copy, but give it a different value.
>>> r4 = r1(value='1K')                     # You can also call the object directly to make copies.
>>> r5, r6, r7 = r1(3)                      # Make 3 copies of the resistor.
>>> r8, r9, r10 = r1(value=[110,220,330])   # Make 3 copies, each with a different value.
>>> r11, r12 = 2 * r1                       # Make copies using the '*' operator.
```

In some cases it's clearer to create parts by copying a *template part* that doesn't actually get included in the netlist for the circuitry. This is done like so:

```
>>> r_template = Part('device', 'R', dest=TEMPLATE)  # Create a resistor just for copying.
>>> r1 = r_template(value='1K')  # Make copy that becomes part of the actual circuitry.
```

4.2.4 Accessing Part Pins and Bus Lines

You can access the pins on a part or the individual nets of a bus using numbers, slices, strings, and regular expressions, either singly or in any combination.

Suppose you have a PIC10 processor in a six-pin package:

```
>>> pic10 = Part('microchip_pic10mcu', 'pic10f220-i/ot')
>>> pic10
PIC10F220-I/OT:
        Pin 1/ICSPDAT/AN0/GP0: BIDIRECTIONAL
        Pin 2/VSS: POWER-IN
        Pin 3/ICSPCLK/AN1/GP1: BIDIRECTIONAL
        Pin 4/T0CKI/FOSC4/GP2: BIDIRECTIONAL
        Pin 5/VDD: POWER-IN
        Pin 6/Vpp/~MCLR~/GP3: INPUT
```

The most natural way to access one of its pins is to give the pin number in brackets:

```
>>> pic10[3]
Pin 3/ICSPCLK/AN1/GP1: BIDIRECTIONAL
```

(If you have a part in a BGA package with pins numbers like `C11`, then you'll have to enter the pin number as a quoted string like `'C11'`.)

You can also get several pins at once in a list:

```
>>> pic10[3,1,6]
[Pin 3/ICSPCLK/AN1/GP1: BIDIRECTIONAL, Pin 1/ICSPDAT/AN0/GP0: BIDIRECTIONAL, Pin 6/Vpp/~MCLR~/GP3: IN
```

You can even use Python slice notation:

```
>>> pic10[2:4]  # Get pins 2 through 4.
[Pin 2/VSS: POWER-IN, Pin 3/ICSPCLK/AN1/GP1: BIDIRECTIONAL, Pin 4/T0CKI/FOSC4/GP2: BIDIRECTIONAL]
>>> pic10[4:2]  # Get pins 4 through 2.
[Pin 4/T0CKI/FOSC4/GP2: BIDIRECTIONAL, Pin 3/ICSPCLK/AN1/GP1: BIDIRECTIONAL, Pin 2/VSS: POWER-IN]
>>> pic10[:]    # Get all the pins.
[Pin 1/ICSPDAT/AN0/GP0: BIDIRECTIONAL, Pin 2/VSS: POWER-IN, Pin 3/ICSPCLK/AN1/GP1: BIDIRECTIONAL, Pi
```

(It's important to note that the slice notation used by SKiDL for parts is slightly different than standard Python. In Python, a slice `n:m` would fetch indices `n`, `n+1`, `...`, `m-1`. With SKiDL, it actually fetches all the way up to the last number: `n`, `n+1`, `...`, `m-1`, `m`. The reason for doing this is that most electronics designers are used to the bounds on a slice including both endpoints. Perhaps it is a mistake to do it this way. We'll see...)

Instead of pin numbers, sometimes it makes the design intent more clear to access pins by their names. For example, it's more obvious that a voltage supply net is being attached to the power pin of the processor when it's expressed like this:

```
pic10['VDD'] += supply_5V
```

You can use multiple names or regular expressions to get more than one pin:

```
>>> pic10['VDD','VSS']
[Pin 5/VDD: POWER-IN, Pin 2/VSS: POWER-IN]
>>> pic10['.*GP[1-3]']
[Pin 3/ICSPCLK/AN1/GP1: BIDIRECTIONAL, Pin 4/T0CKI/FOSC4/GP2: BIDIRECTIONAL, Pin 6/Vpp/~MCLR~/GP3: IN
```

It can be tedious and error prone entering all the quote marks if you're accessing many pin names. SKiDL lets you enter a single, comma-delimited string of pin names:

```
>>> pic10['.*GP0, .*GP1, .*GP2']
[Pin 1/ICSPDAT/AN0/GP0: BIDIRECTIONAL, Pin 3/ICSPCLK/AN1/GP1: BIDIRECTIONAL, Pin 4/T0CKI/FOSC4/GP2: B
```

`Part` objects also provide the `get_pins()` function which can select pins in even more ways. For example, this would get every bidirectional pin of the processor:

```
>>> pic10.get_pins(func=Pin.BIDIR)
[Pin 1/ICSPDAT/AN0/GP0: BIDIRECTIONAL, Pin 3/ICSPCLK/AN1/GP1: BIDIRECTIONAL, Pin 4/T0CKI/FOSC4/GP2: B
```

However, slice notation doesn't work with pin names. You'll get an error if you try that.

Accessing the individual nets of a bus works similarly to accessing part pins:

```
>>> a = Net('NET_A')  # Create a named net.
>>> b = Bus('BUS_B', 8, a)  # Create a nine-bit bus.
>>> b
BUS_B:
        BUS_B0:  # Note how the individual nets of the bus are named.
        BUS_B1:
        BUS_B2:
        BUS_B3:
        BUS_B4:
        BUS_B5:
```

```
        BUS_B6:
        BUS_B7:
        NET_A:   # The last net retains its original name.
>>> b[0]  # Get the first net of the bus.
BUS_B0:
>>> b[4,8]  # Get the fifth and ninth bus lines.
[BUS_B4: , NET_A: ]
>>> b[3:0]  # Get the first four bus lines in reverse order.
[BUS_B3: , BUS_B2: , BUS_B1: , BUS_B0: ]
>>> b['BUS_B.*']  # Get all the bus lines except the last one.
[BUS_B0: , BUS_B1: , BUS_B2: , BUS_B3: , BUS_B4: , BUS_B5: , BUS_B6: , BUS_B7: ]
>>> b['NET_A']  # Get the last bus line.
NET_A:
```

4.2.5 Making Connections

Pins, nets, parts and buses can all be connected together in various ways, but the primary rule of SKiDL connections is:

> **The ‘‘+=‘‘ operator is the only way to make connections!**

At times you'll mistakenly try to make connections using the assignment operator (=). In many cases, SKiDL warns you if you do that, but there are situations where it can't (because Python is a general-purpose programming language where assignment is a necessary operation). So remember the primary rule!

After the primary rule, the next thing to remember is that SKiDL's main purpose is creating netlists. To that end, it handles four basic, connection operations:

Pin-to-Net: A pin is connected to a net, adding it to the list of pins connected to that net. If the pin is already attached to other nets, then those nets are connected to this net as well.

Net-to-Pin: This is the same as doing a pin-to-net connection.

Pin-to-Pin: A net is created and both pins are attached to it. If one or both pins are already connected to other nets, then those nets are connected to the newly-created net as well.

Net-to-Net: Connecting one net to another *merges* the pins on both nets onto a single, larger net.

There are three variants of each connection operation:

One-to-One: This is the most frequent type of connection, for example, connecting one pin to another or connecting a pin to a net.

One-to-Many: This mainly occurs when multiple pins are connected to the same net, like when multiple ground pins of a chip are connected to the circuit ground net.

Many-to-Many: This usually involves bus connections to a part, such as connecting a bus to the data or address pins of a processor. But there must be the same number of things to connect in each set, e.g. you can't connect three pins to four nets.

As a first example, let's connect a net to a pin on a part:

```
>>> pic10 = Part('microchip_pic10mcu','pic10f220-i/ot')  # Get a part.
>>> io = Net('IO_NET')     # Create a net.
>>> pic10['.*GP0'] += io  # Connect the net to a part pin.
>>> io                     # Show the pins connected to the net.
IO_NET: Pin 1/ICSPDAT/AN0/GP0: BIDIRECTIONAL
```

You can do the same operation in reverse by connecting the part pin to the net with the same result:

```
>>> pic10 = Part('microchip_pic10mcu','pic10f220-i/ot')
>>> io = Net('IO_NET')
>>> io += pic10['.*GP0']  # Connect a part pin to the net.
>>> io
IO_NET: Pin 1/ICSPDAT/AN0/GP0: BIDIRECTIONAL
```

You can also connect a pin directly to another pin. In this case, an *implicit net* will be created between the pins that can be accessed using the `net` attribute of either part pin:

```
>>> pic10['.*GP1'] += pic10['.*GP2']  # Connect two pins together.
>>> pic10['.*GP1'].net      # Show the net connected to the pin.
N$1: Pin 4/T0CKI/FOSC4/GP2: BIDIRECTIONAL, Pin 3/ICSPCLK/AN1/GP1: BIDIRECTIONAL
>>> pic10['.*GP2'].net      # Show the net connected to the other pin. Same thing!
N$1: Pin 4/T0CKI/FOSC4/GP2: BIDIRECTIONAL, Pin 3/ICSPCLK/AN1/GP1: BIDIRECTIONAL
```

You can connect multiple pins together, all at once:

```
>>> pic10[1] += pic10[2,3,6]
>>> pic10[1].net
N$1: Pin 6/Vpp/~MCLR~/GP3: INPUT, Pin 3/ICSPCLK/AN1/GP1: BIDIRECTIONAL, Pin 1/ICSPDAT/AN0/GP0: BIDIRE
```

Or you can do it incrementally:

```
>>> pic10[1] += pic10[2]
>>> pic10[1] += pic10[3]
>>> pic10[1] += pic10[6]
>>> pic10[1].net
N$1: Pin 2/VSS: POWER-IN, Pin 6/Vpp/~MCLR~/GP3: INPUT, Pin 1/ICSPDAT/AN0/GP0: BIDIRECTIONAL, Pin 3/IC
```

If you connect pins on separate nets together, then all the pins are merged onto the same net:

```
>>> pic10[1] += pic10[2]  # Put pins 1 & 2 on one net.
>>> pic10[1].net
N$1: Pin 1/ICSPDAT/AN0/GP0: BIDIRECTIONAL, Pin 2/VSS: POWER-IN
>>> pic10[3] += pic10[4]  # Put pins 3 & 4 on another net.
>>> pic10[3].net
N$2: Pin 4/T0CKI/FOSC4/GP2: BIDIRECTIONAL, Pin 3/ICSPCLK/AN1/GP1: BIDIRECTIONAL
>>> pic10[1] += pic10[4]  # Connect two pins from different nets.
>>> pic10[3].net          # Now all the pins are on the same net!
N$2: Pin 1/ICSPDAT/AN0/GP0: BIDIRECTIONAL, Pin 2/VSS: POWER-IN, Pin 4/T0CKI/FOSC4/GP2: BIDIRECTIONAL,
```

Here's an example of connecting a three-bit bus to three pins on a part:

```
>>> pic10 = Part('microchip_pic10mcu','pic10f220-i/ot')
>>> pic10
PIC10F220-I/OT:
        Pin 1/ICSPDAT/AN0/GP0: BIDIRECTIONAL
        Pin 2/VSS: POWER-IN
        Pin 3/ICSPCLK/AN1/GP1: BIDIRECTIONAL
        Pin 4/T0CKI/FOSC4/GP2: BIDIRECTIONAL
        Pin 5/VDD: POWER-IN
        Pin 6/Vpp/~MCLR~/GP3: INPUT
>>> b = Bus('GP', 3)         # Create a 3-bit bus.
>>> pic10[4,3,1] += b[2:0]  # Connect bus to part pins, one-to-one.
>>> b
GP:
        GP0: Pin 1/ICSPDAT/AN0/GP0: BIDIRECTIONAL
        GP1: Pin 3/ICSPCLK/AN1/GP1: BIDIRECTIONAL
        GP2: Pin 4/T0CKI/FOSC4/GP2: BIDIRECTIONAL
```

But SKiDL will warn you if there aren't the same number of things to connect on each side:

```
>>> pic10[4,3,1] += b[1:0]  # Too few bus lines for the pins!
ERROR: Connection mismatch 3 != 2!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "c:\xesscorp\kicad\tools\skidl\skidl\skidl.py", line 2630, in __iadd__
    raise Exception
Exception
```

4.2.6 Hierarchy

SKiDL supports the encapsulation of parts, nets and buses into modules that can be replicated to reduce the design effort, and can be used in other modules to create a functional hierarchy. It does this using Python's built-in machinery for defining and calling functions so there's almost nothing new to learn.

As an example, here's the voltage divider as a module:

```
from skidl import *
import sys

# Define the voltage divider module. The @SubCircuit decorator
# handles some skidl housekeeping that needs to be done.
@SubCircuit
def vdiv(inp, outp):
    """Divide inp voltage by 3 and place it on outp net."""
    rup = Part('device', 'R', value='1K', footprint='Resistors_SMD:R_0805')
    rlo = Part('device','R', value='500', footprint='Resistors_SMD:R_0805')
    rup[1,2] += inp, outp
    rlo[1,2] += outp, gnd

gnd = Net('GND')          # GLobal ground net.
input_net = Net('IN')     # Net with the voltage to be divided.
output_net = Net('OUT')   # Net with the divided voltage.

# Instantiate the voltage divider and connect it to the input & output nets.
vdiv(input_net, output_net)

generate_netlist(sys.stdout)
```

For the most part, `vdiv` is just a standard Python function: it accepts inputs, it performs operations on them, and it could return outputs (but in this case, it doesn't need to). Other than the `@SubCircuit` decorator that appears before the function definition, `vdiv` is just a Python function and it can do anything that a Python function can do.

Here's the netlist that's generated:

```
(export (version D)
  (design
    (source "C:/Users/DEVB/PycharmProjects/test1\test.py")
    (date "08/15/2016 03:35 PM")
    (tool "SKiDL (0.0.1)"))
  (components
    (comp (ref R1)
      (value 1K)
      (footprint Resistors_SMD:R_0805))
    (comp (ref R2)
      (value 500)
      (footprint Resistors_SMD:R_0805)))
```

```
  (nets
    (net (code 0) (name "IN")
      (node (ref R1) (pin 1)))
    (net (code 1) (name "OUT")
      (node (ref R1) (pin 2))
      (node (ref R2) (pin 1)))
    (net (code 2) (name "GND")
      (node (ref R2) (pin 2))))
)
```

For an example of a multi-level hierarchy, the `multi_vdiv` module shown below can use the `vdiv` module to divide a voltage multiple times:

```
from skidl import *
import sys

# Define the voltage divider module.
@SubCircuit
def vdiv(inp, outp):
    """Divide inp voltage by 3 and place it on outp net."""
    rup = Part('device', 'R', value='1K', footprint='Resistors_SMD:R_0805')
    rlo = Part('device','R', value='500', footprint='Resistors_SMD:R_0805')
    rup[1,2] += inp, outp
    rlo[1,2] += outp, gnd

@SubCircuit
def multi_vdiv(repeat, inp, outp):
    """Divide inp voltage by 3 ** repeat and place it on outp net."""
    for _ in range(repeat):
        out_net = Net()      # Create an output net for the current stage.
        vdiv(inp, out_net)   # Instantiate a divider stage.
        inp = out_net        # The output net becomes the input net for the next stage.
    outp += out_net          # Connect the output from the last stage to the module output net.

gnd = Net('GND')          # GLobal ground net.
input_net = Net('IN')     # Net with the voltage to be divided.
output_net = Net('OUT')   # Net with the divided voltage.
multi_vdiv(3, input_net, output_net)  # Run the input through 3 voltage dividers.

generate_netlist(sys.stdout)
```

(For the EE's out there: *yes, I know cascading three simple voltage dividers will not multiplicatively scale the input voltage because of the input and output impedances of each stage!* It's just the simplest example I could think of that shows the feature.)

And here's the resulting netlist:

```
(export (version D)
  (design
    (source "C:/Users/DEVB/PycharmProjects/test1\test.py")
    (date "08/15/2016 05:52 PM")
    (tool "SKiDL (0.0.1)"))
  (components
    (comp (ref R1)
      (value 1K)
      (footprint Resistors_SMD:R_0805))
    (comp (ref R2)
      (value 500)
      (footprint Resistors_SMD:R_0805))
```

```
    (comp (ref R3)
      (value 1K)
      (footprint Resistors_SMD:R_0805))
    (comp (ref R4)
      (value 500)
      (footprint Resistors_SMD:R_0805))
    (comp (ref R5)
      (value 1K)
      (footprint Resistors_SMD:R_0805))
    (comp (ref R6)
      (value 500)
      (footprint Resistors_SMD:R_0805)))
  (nets
    (net (code 0) (name "IN")
      (node (ref R1) (pin 1)))
    (net (code 1) (name "N$1")
      (node (ref R2) (pin 1))
      (node (ref R1) (pin 2))
      (node (ref R3) (pin 1)))
    (net (code 2) (name "GND")
      (node (ref R4) (pin 2))
      (node (ref R6) (pin 2))
      (node (ref R2) (pin 2)))
    (net (code 3) (name "N$2")
      (node (ref R5) (pin 1))
      (node (ref R3) (pin 2))
      (node (ref R4) (pin 1)))
    (net (code 4) (name "OUT")
      (node (ref R5) (pin 2))
      (node (ref R6) (pin 1))))
)
```

4.2.7 Doodads

SKiDL has a few features that don't fit into any other category. Here they are.

No Connects

Sometimes you will use a part, but you won't use every pin. The ERC will complain about those unconnected pins:

```
>>> pic10 = Part('microchip_pic10mcu','pic10f220-i/ot')
>>> ERC()
ERC WARNING: Unconnected pin: BIDIRECTIONAL pin 1/ICSPDAT/AN0/GP0 of PIC10F220-I/OT/IC1.
ERC WARNING: Unconnected pin: POWER-IN pin 2/VSS of PIC10F220-I/OT/IC1.
ERC WARNING: Unconnected pin: BIDIRECTIONAL pin 3/ICSPCLK/AN1/GP1 of PIC10F220-I/OT/IC1.
ERC WARNING: Unconnected pin: BIDIRECTIONAL pin 4/T0CKI/FOSC4/GP2 of PIC10F220-I/OT/IC1.
ERC WARNING: Unconnected pin: POWER-IN pin 5/VDD of PIC10F220-I/OT/IC1.
ERC WARNING: Unconnected pin: INPUT pin 6/Vpp/~MCLR~/GP3 of PIC10F220-I/OT/IC1.
```

If you have pins that you intentionally want to leave unconnected, then attach them to the special-purpose `NC` (no-connect) net and the warnings will be supressed:

```
>>> pic10[1,3,4] += NC
>>> ERC()
ERC WARNING: Unconnected pin: POWER-IN pin 2/VSS of PIC10F220-I/OT/IC1.
```

```
ERC WARNING: Unconnected pin: POWER-IN pin 5/VDD of PIC10F220-I/OT/IC1.
ERC WARNING: Unconnected pin: INPUT pin 6/Vpp/~MCLR~/GP3 of PIC10F220-I/OT/IC1.
```

In fact, if you have a part with many pins that are not going to be used, you can start off by attaching all the pins to the NC net. After that, you can attach the pins you're using to normal nets and they will be removed from the NC net:

```
my_part[:] += NC  # Connect every pin to NC net.
...
my_part[5] += Net()  # Pin 5 is no longer unconnected.
```

The NC net is the only net for which this happens. For all other nets, connecting two or more nets to the same pin merges those nets and all the pins on them together.

Net Drive Level

Certain parts have power pins that are required to be driven by a power supply net or else ERC warnings ensue. This condition is usually satisfied if the power pins are driven by the output of another part like a voltage regulator. But if the regulator output passes through something like a ferrite bead (to remove noise), then the filtered signal is no longer a supply net and an ERC warning is issued.

In order to satisfy the ERC, the drive strength of a net can be set manually using its drive attribute. As a simple example, consider connecting a net to the power supply input of a processor and then running the ERC:

```
>>> pic10 = Part('microchip_pic10mcu','pic10f220-i/ot')
>>> a = Net()
>>> pic10['VDD'] += a
>>> ERC()
...
ERC WARNING: Insufficient drive current on net N$1 for pin POWER-IN pin 5/VDD of PIC10F220-I/OT/IC1
...
```

This issue is fixed by changing the drive attribute of the net:

```
>>> pic10 = Part('microchip_pic10mcu','pic10f220-i/ot')
>>> a = Net()
>>> pic10['VDD'] += a
>>> a.drive = POWER
>>> ERC()
...
(Insufficient drive warning is no longer present.)
...
```

You can set the drive attribute at any time to any defined level, but POWER is probably the only setting you'll use. Also, the drive attribute retains the highest of all the levels it has been set at, so once it is set to the POWER level it is impossible to set it to a lower level. (This is done during internal processing to keep a net at the highest drive level of any of the pins that have been attached to it.)

In short, for any net you create that supplies power to devices in your circuit, you should probably set its drive attribute to POWER. This is equivalent to attaching power flags to nets in some ECAD packages like KiCad.

Selectively Supressing ERC Messages

Sometimes a portion of your circuit throws a lot of ERC warnings or errors even though you know it's correct. SKiDL provides flags that allow you to turn off the ERC for selected nets, pins, and parts like so:

```
my_net.do_erc = False       # Turns of ERC for this particular net.
my_part[5].do_erc = False   # Turns off ERC for this pin of this part.
my_part.do_erc = False      # Turns off ERC for all the pins of this part.
```

4.3 Converting Existing Designs to SKiDL

If you have an existing schematic-based design, you can convert it to SKiDL as follows:

1. Generate a netlist file for your design using whatever procedure your ECAD system provides. For this discussion, call the netlist file `my_design.net`.
2. Convert the netlist file into a SKiDL program using the following command:

```
    netlist_to_skidl -i my_design.net -o my_design.py -w
```

That's it! You can execute the `my_design.py` script and it will regenerate the netlist. Or you can use the script as a subcircuit in a larger design. Or do anything else that a SKiDL-based design supports.

CHAPTER 5

API

SKiDL: A Python-Based Schematic Design Language

This module extends Python with the ability to design electronic circuits. It provides classes for working with **1)** electronic parts (`Part`), **2)** collections of part terminals (`Pin`) connected via wires (`Net`), and **3)** groups of related nets (`Bus`). Using these classes, you can concisely describe the interconnection of components using a linear and/or hierarchical structure. It also provides the capability to check the resulting circuitry for the violation of electrical rules. The output of a SKiDL-enabled Python script is a netlist that can be imported into a PCB layout tool.

class `skidl.skidl.`**`Alias`**(*name*, *id_tag=None*)

An alias can be added to another object to give it another name. Since an object might have several aliases, each alias can be tagged with an identifier to discriminate between them.

Parameters

- **`name`** – The alias name.
- **`id_tag`** – The identifier tag.

`__eq__`(*search*)

Return true if one alias is equal to another.

The aliases are equal if the following conditions are both true:

```
1. The ids must match or one or both ids must be something
    that evaluates to False (i.e., None, empty string or list, etc.).

2. The names must match based on using one name as a
    regular expression to compare to the other.
```

Parameters **`search`** – The Alias object which self will be compared to.

class `skidl.skidl.`**`Bus`**(*name*, **args*, ***attribs*)

This class collects one or more nets into a group that can be indexed.

Parameters

- **`name`** – A string with the name of the bus.
- **`args`** – A list of ints, pins, nets, buses to attach to the net.

Keyword Arguments **attribs** – A dictionary of attributes and values to attach to the Net object.

Example

```
n = Net()
led1 = Part('device', 'LED')
b = Bus('B', 8, n, led1['K'])
```

__call__(*num_copies=1*, ***attribs*)
Make zero or more copies of this bus.

Parameters **num_copies** – Number of copies to make of this bus.

Keyword Arguments **attribs** – Name/value pairs for setting attributes for the copy.

Returns A list of Bus copies or a Bus if num_copies==1.

Raises Exception if the requested number of copies is a non-integer or negative.

Notes

An instance of a bus can be copied just by calling it like so:

```
b = Bus('A', 8)  # Create a bus.
b_copy = b(2)    # Get two copies of the bus.
```

You can also use the multiplication operator to make copies:

```
b = 10 * Bus('A', 8)  # Create an array of buses.
```

__getitem__(**ids*)
Return a bus made up of the nets at the given indices.

Parameters **ids** – A list of indices of bus lines. These can be individual numbers, net names, nested lists, or slices.

Returns A bus if the indices are valid, otherwise None.

__iadd__(**pins_nets_buses*)
Return the bus after connecting one or more nets, pins, or buses.

Parameters **pins_nets_buses** – One or more Pin, Net or Bus objects or lists/tuples of them.

Returns The updated bus with the new connections.

Notes

You can connect nets or pins to a bus like so:

```
p = Pin()        # Create a pin.
n = Net()        # Create a net.
b = Bus('B', 2) # Create a two-wire bus.
b += p,n         # Connect pin and net to B[0] and B[1].
```

__len__()
Return the number of nets in this bus.

__mul__(*num_copies=1*, ***attribs*)
Make zero or more copies of this bus.

Parameters **num_copies** – Number of copies to make of this bus.

Keyword Arguments **attribs** – Name/value pairs for setting attributes for the copy.

Returns A list of Bus copies or a Bus if num_copies==1.

Raises Exception if the requested number of copies is a non-integer or negative.

Notes

An instance of a bus can be copied just by calling it like so:

```
b = Bus('A', 8)  # Create a bus.
b_copy = b(2)    # Get two copies of the bus.
```

You can also use the multiplication operator to make copies:

```
b = 10 * Bus('A', 8)  # Create an array of buses.
```

__repr__()
Return a list of the nets in this bus as a string.

__rmul__(*num_copies=1*, ***attribs*)
Make zero or more copies of this bus.

Parameters **num_copies** – Number of copies to make of this bus.

Keyword Arguments **attribs** – Name/value pairs for setting attributes for the copy.

Returns A list of Bus copies or a Bus if num_copies==1.

Raises Exception if the requested number of copies is a non-integer or negative.

Notes

An instance of a bus can be copied just by calling it like so:

```
b = Bus('A', 8)  # Create a bus.
b_copy = b(2)    # Get two copies of the bus.
```

You can also use the multiplication operator to make copies:

```
b = 10 * Bus('A', 8)  # Create an array of buses.
```

__setitem__(*ids*, **pins_nets_buses*)
You can't assign to bus lines. You must use the += operator.

This method is a work-around that allows the use of the += for making connections to bus lines while prohibiting direct assignment. Python processes something like my_bus[7:0] += 8 * Pin() as follows:

```
1. Part.__getitem__ is called with '7:0' as the index. This
   returns a NetPinList of eight nets from my_bus.
2. The NetPinList.__iadd__ method is passed the NetPinList and
   the thing to connect to the it (eight pins in this case). This
   method makes the actual connection to the part pin or pins. Then
   it creates an iadd_flag attribute in the object it returns.
3. Finally, Bus.__setitem__ is called. If the iadd_flag attribute
   is true in the passed argument, then __setitem__ was entered
   as part of processing the += operator. If there is no
   iadd_flag attribute, then __setitem__ was entered as a result
   of using a direct assignment, which is not allowed.
```

__str__()
Return a list of the nets in this bus as a string.

connect(**pins_nets_buses*)
Return the bus after connecting one or more nets, pins, or buses.

Parameters **pins_nets_buses** – One or more Pin, Net or Bus objects or lists/tuples of them.

Returns The updated bus with the new connections.

Notes

You can connect nets or pins to a bus like so:

```
p = Pin()        # Create a pin.
n = Net()        # Create a net.
b = Bus('B', 2)  # Create a two-wire bus.
b += p,n         # Connect pin and net to B[0] and B[1].
```

copy(*num_copies=1*, ***attribs*)
Make zero or more copies of this bus.

Parameters **num_copies** – Number of copies to make of this bus.

Keyword Arguments **attribs** – Name/value pairs for setting attributes for the copy.

Returns A list of Bus copies or a Bus if num_copies==1.

Raises Exception if the requested number of copies is a non-integer or negative.

Notes

An instance of a bus can be copied just by calling it like so:

```
b = Bus('A', 8)  # Create a bus.
b_copy = b(2)    # Get two copies of the bus.
```

You can also use the multiplication operator to make copies:

```
b = 10 * Bus('A', 8)  # Create an array of buses.
```

name
Get, set and delete the name of the bus.

When setting the bus name, if another bus with the same name is found, the name for this bus is adjusted to make it unique.

skidl.skidl.**Circuit**
alias of *SubCircuit*

class skidl.skidl.**Net**(*name=None*, **pins_nets_buses*, ***attribs*)
Lists of connected pins are stored as nets using this class.

Parameters

- **name** – A string with the name of the net. If None or '', then a unique net name will be assigned.
- ***pins_nets_buses** – One or more Pin, Net, or Bus objects or lists/tuples of them to be connected to this net.

Keyword Arguments **attribs** – A dictionary of attributes and values to attach to the Net object.

__call__(*num_copies=1*, ***attribs*)
Make zero or more copies of this net.

Parameters **num_copies** – Number of copies to make of this net.

Keyword Arguments **attribs** – Name/value pairs for setting attributes for the copy.

Returns A list of Net copies or a Net if num_copies==1.

Raises Exception if the requested number of copies is a non-integer or negative.

Notes

An instance of a net can be copied just by calling it like so:

```
n = Net('A')     # Create a net.
n_copy = n()     # Copy the net.
```

You can also use the multiplication operator to make copies:

```
n = 10 * Net('A')  # Create an array of nets.
```

__iadd__(**pins_nets_buses*)
Return the net after connecting other pins, nets, and buses to it.

Parameters ***pins_nets_buses** – One or more Pin, Net, or Bus objects or lists/tuples of them to be connected to this net.

Returns The updated net with the new connections.

Notes

Connections to nets can also be made using the += operator like so:

```
atmega = Part('atmel', 'ATMEGA16U2')
net = Net()
net += atmega[1]  # Connects pin 1 of chip to the net.
```

__len__()
Return the number of pins attached to this net.

__mul__(*num_copies=1*, ***attribs*)
Make zero or more copies of this net.

Parameters **num_copies** – Number of copies to make of this net.

Keyword Arguments **attribs** – Name/value pairs for setting attributes for the copy.

Returns A list of Net copies or a Net if num_copies==1.

Raises Exception if the requested number of copies is a non-integer or negative.

Notes

An instance of a net can be copied just by calling it like so:

```
n = Net('A')     # Create a net.
n_copy = n()     # Copy the net.
```

You can also use the multiplication operator to make copies:

```
n = 10 * Net('A')  # Create an array of nets.
```

__repr__()
Return a list of the pins on this net as a string.

__rmul__(*num_copies=1*, ***attribs*)
Make zero or more copies of this net.

Parameters **num_copies** – Number of copies to make of this net.

Keyword Arguments **attribs** – Name/value pairs for setting attributes for the copy.

Returns A list of Net copies or a Net if num_copies==1.

Raises Exception if the requested number of copies is a non-integer or negative.

Notes

An instance of a net can be copied just by calling it like so:

```
n = Net('A')     # Create a net.
n_copy = n()     # Copy the net.
```

You can also use the multiplication operator to make copies:

```
n = 10 * Net('A')  # Create an array of nets.
```

__str__()
Return a list of the pins on this net as a string.

connect(**pins_nets_buses*)
Return the net after connecting other pins, nets, and buses to it.

Parameters ***pins_nets_buses** – One or more Pin, Net, or Bus objects or lists/tuples of them to be connected to this net.

Returns The updated net with the new connections.

Notes

Connections to nets can also be made using the += operator like so:

```
atmega = Part('atmel', 'ATMEGA16U2')
net = Net()
net += atmega[1]  # Connects pin 1 of chip to the net.
```

copy(*num_copies=1*, ***attribs*)
Make zero or more copies of this net.

Parameters **num_copies** – Number of copies to make of this net.

Keyword Arguments **attribs** – Name/value pairs for setting attributes for the copy.

Returns A list of Net copies or a Net if num_copies==1.

Raises Exception if the requested number of copies is a non-integer or negative.

Notes

An instance of a net can be copied just by calling it like so:

```
n = Net('A')     # Create a net.
n_copy = n()     # Copy the net.
```

You can also use the multiplication operator to make copies:

```
n = 10 * Net('A')  # Create an array of nets.
```

drive
Get, set and delete the drive strength of this net.

The drive strength cannot be set to a value less than its current value. So as pins are added to a net, the drive strength reflects the maximum drive value of the pins currently on the net.

name
Get, set and delete the name of this net.

When setting the net name, if another net with the same name is found, the name for this net is adjusted to make it unique.

class skidl.skidl.**Part**(*lib=None*, *name=None*, *dest='NETLIST'*, *tool='kicad'*, *connections=None*, *part_defn=None*, ***attribs*)
A class for storing a definition of a schematic part.

ref
String storing the reference of a part within a schematic (e.g., 'R5').

value
String storing the part value (e.g., '3K3').

footprint
String storing the PCB footprint associated with a part (e.g., SOIC-8).

pins
List of Pin objects for this part.

Parameters

- **lib** – Either a SchLib object or a schematic part library file name.
- **name** – A string with name of the part to find in the library, or to assign to the part defined by the part definition.
- **part_defn** – A list of strings that define the part (usually read from a schematic library file).
- **tool** – The format for the library file or part definition (e.g., KICAD).
- **dest** – String that indicates where the part is destined for (e.g., LIBRARY).
- **connections** – A dictionary with part pin names/numbers as keys and the names of nets to which they will be connected as values. For example: { 'IN-':'a_in', 'IN+':'GND', '1':'AMPED_OUTPUT', '14':'VCC', '7':'GND' }

Keyword Arguments **attribs** – Name/value pairs for setting attributes for the part. For example, manf_num='LM4808MP-8' would create an attribute named 'manf_num' for the part and assign it the value 'LM4808MP-8'.

Raises

- * Exception if the part library and definition are both missing.

- * Exception if an unknown file format is requested.

__call__(*num_copies=1*, *dest='NETLIST'*, ***attribs*)
Make zero or more copies of this part while maintaining all pin/net connections.

Parameters

- **num_copies** – Number of copies to make of this part.
- **dest** – Indicates where the copy is destined for (e.g., NETLIST).

Keyword Arguments **attribs** – Name/value pairs for setting attributes for the copy.

Returns A list of Part copies or a single Part if num_copies==1.

Raises Exception if the requested number of copies is a non-integer or negative.

Notes

An instance of a part can be copied just by calling it like so:

```
res = Part('device','R')     # Get a resistor.
res_copy = res(value='1K')   # Copy the resistor and set resistance value.
```

You can also use the multiplication operator to make copies:

```
cap = Part('device', 'C')    # Get a capacitor
caps = 10 * cap              # Make an array with 10 copies of it.
```

__getitem__(**pin_ids*, ***criteria*)
Return list of part pins selected by pin numbers or names.

Parameters **pin_ids** – A list of strings containing pin names, numbers, regular expressions, slices, lists or tuples. If empty, then it will select all pins.

Keyword Arguments **criteria** – Key/value pairs that specify attribute values the pins must have in order to be selected.

Returns A list of pins matching the given IDs and satisfying all the criteria, or just a single Pin object if only a single match was found. Or None if no match was found.

Notes

Pins can be selected from a part by using brackets like so:

```
atmega = Part('atmel', 'ATMEGA16U2')
net = Net()
atmega[1] += net  # Connects pin 1 of chip to the net.
net += atmega['.*RESET.*']  # Connects reset pin to the net.
```

__mul__(*num_copies=1*, *dest='NETLIST'*, ***attribs*)
Make zero or more copies of this part while maintaining all pin/net connections.

Parameters

- **num_copies** – Number of copies to make of this part.
- **dest** – Indicates where the copy is destined for (e.g., NETLIST).

Keyword Arguments **attribs** – Name/value pairs for setting attributes for the copy.

Returns A list of Part copies or a single Part if num_copies==1.

Raises Exception if the requested number of copies is a non-integer or negative.

Notes

An instance of a part can be copied just by calling it like so:

```
res = Part('device','R')     # Get a resistor.
res_copy = res(value='1K')   # Copy the resistor and set resistance value.
```

You can also use the multiplication operator to make copies:

```
cap = Part('device', 'C')    # Get a capacitor
caps = 10 * cap              # Make an array with 10 copies of it.
```

__repr__()
Return a description of the pins on this part as a string.

__rmul__(*num_copies=1*, *dest='NETLIST'*, ***attribs*)
Make zero or more copies of this part while maintaining all pin/net connections.

Parameters

- **num_copies** – Number of copies to make of this part.
- **dest** – Indicates where the copy is destined for (e.g., NETLIST).

Keyword Arguments **attribs** – Name/value pairs for setting attributes for the copy.

Returns A list of Part copies or a single Part if num_copies==1.

Raises Exception if the requested number of copies is a non-integer or negative.

Notes

An instance of a part can be copied just by calling it like so:

```
res = Part('device','R')     # Get a resistor.
res_copy = res(value='1K')   # Copy the resistor and set resistance value.
```

You can also use the multiplication operator to make copies:

```
cap = Part('device', 'C')    # Get a capacitor
caps = 10 * cap              # Make an array with 10 copies of it.
```

__setitem__(*ids*, **pins_nets_buses*)
You can't assign to the pins of parts. You must use the += operator.

This method is a work-around that allows the use of the += for making connections to pins while prohibiting direct assignment. Python processes something like my_part['GND'] += gnd as follows:

```
1. Part.__getitem__ is called with 'GND' as the index. This
   returns a single Pin or a NetPinList.
2. The Pin.__iadd__ or NetPinList.__iadd__ method is passed
   the thing to connect to the pin (gnd in this case). This method
   makes the actual connection to the part pin or pins. Then it
   creates an iadd_flag attribute in the object it returns.
3. Finally, Part.__setitem__ is called. If the iadd_flag attribute
   is true in the passed argument, then __setitem__ was entered
```

```
        as part of processing the += operator. If there is no
        iadd_flag attribute, then __setitem__ was entered as a result
        of using a direct assignment, which is not allowed.
```

__str__()
Return a description of the pins on this part as a string.

copy(*num_copies=1*, *dest='NETLIST'*, ***attribs*)
Make zero or more copies of this part while maintaining all pin/net connections.

Parameters

- **num_copies** – Number of copies to make of this part.
- **dest** – Indicates where the copy is destined for (e.g., NETLIST).

Keyword Arguments **attribs** – Name/value pairs for setting attributes for the copy.

Returns A list of Part copies or a single Part if num_copies==1.

Raises Exception if the requested number of copies is a non-integer or negative.

Notes

An instance of a part can be copied just by calling it like so:

```
res = Part('device','R')     # Get a resistor.
res_copy = res(value='1K')   # Copy the resistor and set resistance value.
```

You can also use the multiplication operator to make copies:

```
cap = Part('device', 'C')    # Get a capacitor
caps = 10 * cap              # Make an array with 10 copies of it.
```

foot
Get, set and delete the part footprint.

get_pins(**pin_ids*, ***criteria*)
Return list of part pins selected by pin numbers or names.

Parameters **pin_ids** – A list of strings containing pin names, numbers, regular expressions, slices, lists or tuples. If empty, then it will select all pins.

Keyword Arguments **criteria** – Key/value pairs that specify attribute values the pins must have in order to be selected.

Returns A list of pins matching the given IDs and satisfying all the criteria, or just a single Pin object if only a single match was found. Or None if no match was found.

Notes

Pins can be selected from a part by using brackets like so:

```
atmega = Part('atmel', 'ATMEGA16U2')
net = Net()
atmega[1] += net  # Connects pin 1 of chip to the net.
net += atmega['.*RESET.*']  # Connects reset pin to the net.
```

make_unit(*label*, **pin_ids*, ***criteria*)
Create a PartUnit from a set of pins in a Part object.

Parts can be organized into smaller pieces called PartUnits. A PartUnit acts like a Part but contains only a subset of the pins of the Part.

Parameters

- **label** – The label used to identify the PartUnit.
- **pin_ids** – A list of strings containing pin names, numbers, regular expressions, slices, lists or tuples.

Keyword Arguments **criteria** – Key/value pairs that specify attribute values the pin must have in order to be selected.

Returns The PartUnit.

ref
Get, set and delete the part reference.

When setting the part reference, if another part with the same reference is found, the reference for this part is adjusted to make it unique.

value
Get, set and delete the part value.

class skidl.skidl.**PartUnit**(*part*, **pin_ids*, ***criteria*)
Create a PartUnit from a set of pins in a Part object.

Parts can be organized into smaller pieces called PartUnits. A PartUnit acts like a Part but contains only a subset of the pins of the Part.

Parameters

- **part** – This is the parent Part whose pins the PartUnit is built from.
- **pin_ids** – A list of strings containing pin names, numbers, regular expressions, slices, lists or tuples.

Keyword Arguments **criteria** – Key/value pairs that specify attribute values the pin must have in order to be selected.

Examples

This will return unit 1 from a part:

```
lm358 = Part('linear','lm358')
lm358a = PartUnit(lm358, unit=1)
```

Or you can specify the pins directly:

```
lm358a = PartUnit(lm358, 1, 2, 3)
```

class skidl.skidl.**Pin**(***attribs*)
A class for storing data about pins for a part.

Parameters **attribs** – Key/value pairs of attributes to add to the library.

nets
The electrical nets this pin is connected to (can be >1).

part
Link to the Part object this pin belongs to.

do_erc
When false, the pin is not checked for ERC violations.

__call__(*num_copies=1*, ***attribs*)
Return copy or list of copies of a pin including any net connection.

Parameters **num_copies** – Number of copies to make of pin.

Keyword Arguments **attribs** – Name/value pairs for setting attributes for the pin.

Notes

An instance of a pin can be copied just by calling it like so:

```
p = Pin()      # Create a pin.
p_copy = p()   # This is a copy of the pin.
```

__iadd__(**pins_nets_buses*)
Return the pin after connecting it to one or more nets or pins.

Parameters **pins_nets_buses** – One or more Pin, Net or Bus objects or lists/tuples of them.

Returns The updated pin with the new connections.

Notes

You can connect nets or pins to a pin like so:

```
p = Pin()      # Create a pin.
n = Net()      # Create a net.
p += net       # Connect the net to the pin.
```

__mul__(*num_copies=1*, ***attribs*)
Return copy or list of copies of a pin including any net connection.

Parameters **num_copies** – Number of copies to make of pin.

Keyword Arguments **attribs** – Name/value pairs for setting attributes for the pin.

Notes

An instance of a pin can be copied just by calling it like so:

```
p = Pin()      # Create a pin.
p_copy = p()   # This is a copy of the pin.
```

__repr__()
Return a description of this pin as a string.

__rmul__(*num_copies=1*, ***attribs*)
Return copy or list of copies of a pin including any net connection.

Parameters **num_copies** – Number of copies to make of pin.

Keyword Arguments **attribs** – Name/value pairs for setting attributes for the pin.

Notes

An instance of a pin can be copied just by calling it like so:

```
p = Pin()      # Create a pin.
p_copy = p()   # This is a copy of the pin.
```

__str__()
Return a description of this pin as a string.

connect(**pins_nets_buses*)
Return the pin after connecting it to one or more nets or pins.

Parameters **pins_nets_buses** – One or more Pin, Net or Bus objects or lists/tuples of them.

Returns The updated pin with the new connections.

Notes

You can connect nets or pins to a pin like so:

```
p = Pin()      # Create a pin.
n = Net()      # Create a net.
p += net       # Connect the net to the pin.
```

copy(*num_copies=1*, ***attribs*)
Return copy or list of copies of a pin including any net connection.

Parameters **num_copies** – Number of copies to make of pin.

Keyword Arguments **attribs** – Name/value pairs for setting attributes for the pin.

Notes

An instance of a pin can be copied just by calling it like so:

```
p = Pin()      # Create a pin.
p_copy = p()   # This is a copy of the pin.
```

net
Return one of the nets the pin is connected to.

class skidl.skidl.**SubCircuit**(*circuit_func*)
Class object that holds the entire netlist of parts and nets. This is initialized once when the module is first imported and then all parts and nets are added to its static members.

parts
List of all the schematic parts as Part objects.

nets
List of all the schematic nets as Net objects.

hierarchy
A '.'-separated concatenation of the names of nested SubCircuits at the current time it is read.

level
The current level in the schematic hierarchy.

context
Stack of contexts for each level in the hierarchy.

circuit_func
The function that creates a given subcircuit.

__call__(**args*, ***kwargs*)
This method is called when you invoke the SubCircuit object to create some schematic circuitry.

classmethod set_pin_conflict_rule(*pin1_func*, *pin2_func*, *conflict_level*)
Set the level of conflict for two types of pins on the same net.

Parameters

- **pin1_func** – The function of the first pin (e.g., Pin.OUTPUT).
- **pin2_func** – The function of the second pin (e.g., Pin.TRISTATE).
- **conflict_level** – Severity of conflict (e.g., cls.OK, cls.WARNING, cls.ERROR).

skidl.skidl.**search**(*term*)
Print a list of components with the regex term within their name, alias, description or keywords.

skidl.skidl.**show**(*lib_name*, *part_name*)
Print the I/O pins for a given part in a library.

CHAPTER 6

Credits

6.1 Development Lead

- XESS Corp. <info@xess.com>

6.2 Contributors

None yet. Why not be the first?

CHAPTER 7

Acknowledgements

SKiDL was inspired by two other projects:

- PHDL is a schematic design language that exemplifies the main features I wanted in SKiDL: concise, text-based design entry with support for hierarchy.
- MyHDL showed how to use the features of Python to support a particular application (designing/simulating digital logic systems) while keeping access to a rich software ecosystem.

CHAPTER 8

History

8.1 0.0.8 (2017-01-11)

- skidl_to_netlist now uses templates.
- Default operation of search() is now less exacting.
- Traceback is now suppressed if show() is passed a part name not in a library.

8.2 0.0.7 (2016-09-11)

- Lack of KISYSMOD environment variable no longer causes an exception.
- requirements.txt file now references the requirements from setup.py.
- Changed setup so it generates a pckg_info file with version, author, email.

8.3 0.0.6 (2016-09-10)

- Fixed error caused when trying to find script name when SKiDL is run in interactive mode.
- Silenced errors/warnings when loading KiCad part description (.dcm) files.

8.4 0.0.5 (2016-09-07)

- SKiDL now searches for parts with a user-configurable list of library search paths.
- Part descriptions and keywords are now loaded from the .dcm file associated with a .lib file.

8.5 0.0.4 (2016-08-27)

- SKiDL scripts can now output netlists in XML format.

8.6 0.0.3 (2016-08-25)

- Added command-line utility to convert netlists into SKiDL programs.

8.7 0.0.2 (2016-08-17)

- Changed the link to the documentation.

8.8 0.0.1 (2016-08-16)

- First release on PyPI.

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

S

Symbols

A

B

C

D

F

G

H

L

M

N

P

R

S

V