circuit

Release 2.0.1

Reity LLC

Mar 29, 2023

CONTENTS

1 Installation and Usage **3**
- 1.1 Examples . . . 3

2 Development **5**
- 2.1 Documentation . . . 5
- 2.2 Testing and Conventions . . . 5
- 2.3 Contributions . . . 6
- 2.4 Versioning . . . 6
- 2.5 Publishing . . . 6
  - 2.5.1 circuit module . . . 6

Python Module Index **27**

Index **29**

Pure-Python library for building and working with logical circuits.

CHAPTER

ONE

INSTALLATION AND USAGE

This library is available as a package on PyPI:

```
python -m pip install circuit
```

The library can be imported in the usual way:

```
import circuit
from circuit import *
```

1.1 Examples

This library makes it possible to programmatically construct logical circuits consisting of interconnected logic gates. The functions corresponding to individual logic gates are represented using the logical library. In the example below, a simple conjunction circuit is constructed, and its input and output gates (corresponding to the logical unary identity function) are created and designated as such:

```
>>> from circuit import circuit, op
>>> c = circuit()
>>> g0 = c.gate(op.id_, is_input=True)
>>> g1 = c.gate(op.id_, is_input=True)
>>> g2 = c.gate(op.and_, [g0, g1])
>>> g3 = c.gate(op.id_, [g2], is_output=True)
>>> c.count() # Number of gates in the circuit.
4
```

The gate list associated with a circuit can be converted into a concise human-readable format, enabling manual inspection of the circuit:

```
>>> c.gates.to_legible()
(('id',), ('id',), ('and', 0, 1), ('id', 2))
```

The circuit accepts two input bits (represented as integers) and can be evaluated on any list of two bits using the `evaluate` method. The result is a bit vector that includes one bit for each output gate:

```
>>> c.evaluate([0, 1])
[0]
>>> [list(c.evaluate(bs)) for bs in [[0, 0], [0, 1], [1, 0], [1, 1]]]
[[0], [0], [0], [1]]
```

Note that the order of the output bits corresponds to the order in which the output gates were originally introduced using the `gate` method. It is possible to specify the signature of a circuit (*i.e.*, the organization of input gates and output gates into distinct bit vectors of specific lengths) at the time the circuit object is created:

```
>>> from circuit import signature
>>> c = circuit(signature([2], [1]))
>>> g0 = c.gate(op.id_, is_input=True)
>>> g1 = c.gate(op.id_, is_input=True)
>>> g2 = c.gate(op.not_, [g0])
>>> g3 = c.gate(op.not_, [g1])
>>> g4 = c.gate(op.xor_, [g2, g3])
>>> g5 = c.gate(op.not_, [g4])
>>> g6 = c.gate(op.id_, [g4], is_output=True)
>>> [list(c.evaluate([bs])) for bs in [[0, 0], [0, 1], [1, 0], [1, 1]]]
[[[0]], [[1]], [[1]], [[0]]]
```

It is also possible to remove all internal gates from which an output gate cannot be reached (such as `g5` in the example above). Doing so does not change the order of the input gates or the order of the output gates:

```
>>> c.count()
7
>>> c.prune_and_topological_sort_stable()
>>> c.count()
6
```

Other methods make it possible to discard a gate, to replace a collection of gates, and to convert a circuit into a boolean function. Descriptions and examples of these and other methods can be found in the documentation for the main library module.

CHAPTER

TWO

DEVELOPMENT

All installation and development dependencies are fully specified in `pyproject.toml`. The `project.optional-dependencies` object is used to specify optional requirements for various development tasks. This makes it possible to specify additional options (such as `docs`, `lint`, and so on) when performing installation using pip:

```
python -m pip install .[docs,lint]
```

2.1 Documentation

The documentation can be generated automatically from the source files using Sphinx:

```
python -m pip install .[docs]
cd docs
sphinx-apidoc -f -E --templatedir=_templates -o _source .. && make html
```

2.2 Testing and Conventions

All unit tests are executed and their coverage is measured when using pytest (see the `pyproject.toml` file for configuration details):

```
python -m pip install .[test]
python -m pytest
```

Alternatively, all unit tests are included in the module itself and can be executed using doctest:

```
python src/circuit/circuit.py -v
```

Style conventions are enforced using Pylint:

```
python -m pip install .[lint]
python -m pylint src/circuit
```

2.3 Contributions

In order to contribute to the source code, open an issue or submit a pull request on the GitHub page page for this library.

2.4 Versioning

Beginning with version 0.2.0, the version number format for this library and the changes to the library associated with version number increments conform with Semantic Versioning 2.0.0.

2.5 Publishing

This library can be published as a package on PyPI by a package maintainer. First, install the dependencies required for packaging and publishing:

```
python -m pip install .[publish]
```

Ensure that the correct version number appears in `pyproject.toml`, and that any links in this README document to the Read the Docs documentation of this package (or its dependencies) have appropriate version numbers. Also ensure that the Read the Docs project for this library has an automation rule that activates and sets as the default all tagged versions. Create and push a tag for this version (replacing `?.?.?` with the version number):

```
git tag ?.?.?
git push origin ?.?.?
```

Remove any old build/distribution files. Then, package the source into a distribution archive using the wheel package:

```
rm -rf build dist src/*.egg-info
python -m build --sdist --wheel .
```

Finally, upload the package distribution archive to PyPI using the twine package:

```
python -m twine upload dist/*
```

2.5.1 circuit module

Pure-Python library for building and working with logical circuits (both as expressions and as graphs).

This library makes it possible to construct logical circuits programmatically by building them up from individual gates.

```
>>> c = circuit()
>>> g0 = c.gate(op.id_, is_input=True)
>>> g1 = c.gate(op.id_, is_input=True)
>>> g2 = c.gate(op.and_, [g0, g1])
>>> g3 = c.gate(op.id_, [g2], is_output=True)
>>> c.count()
4
```

The gate list associated with a circuit can be converted into a concise human-readable format using the *gates.to_legible* method, enabling manual inspection of the circuit.

```
>>> c.gates.to_legible()
(('id',), ('id',), ('and', 0, 1), ('id', 2))
```

A *circuit* object can be evaluated on any list of bits using the *evaluate* method. The result is a bit vector that includes one bit for each output gate.

```
>>> [list(c.evaluate(bs)) for bs in [[0, 0], [0, 1], [1, 0], [1, 1]]]
[[0], [0], [0], [1]]
```

Please refer to the documentation for the *circuit* class for more details on usage, features, and available methods.

circuit.circuit.**operation**
alias of logical.logical.logical

circuit.circuit.**op**
alias of logical.logical.logical

class circuit.circuit.**gate**(*operation=None*, *inputs=None*, *outputs=None*, *is_input=False*, *is_output=False*)
Bases: object

Data structure for an individual circuit logic gate, with attributes that indicate the logical operation corresponding to the gate (represented using an instance of the logical class that is defined in the logical library), the other gate connected gate instances, and whether the gate is designated as an input and/or output gate of the overall circuit to which it belongs.

Parameters

- **operation** (Optional[logical]) – Logical operation that the gate represents.
- **inputs** (Optional[Sequence[Optional[*gate*]]]) – List of input gate object references.
- **outputs** (Optional[Sequence[*gate*]]) – List of output gate object references.
- **is_input** (bool) – Flag indicating if this is an input gate for a circuit.
- **is_output** (bool) – Flag indicating if this is an output gate for a circuit.

```
>>> g0 = gate(op.id_, [])
>>> g1 = gate(op.not_, [])
>>> g2 = gate(op.and_, [g0, g1])
```

The list of inputs, if specified, must have either no entries or a number of entries that matches the operation arity. Otherwise, an exception is raised.

```
>>> g3 = gate(op.and_, [g2])
Traceback (most recent call last):
  ...
ValueError: number of inputs must equal operation arity or zero
```

output(*other*)
Designate another gate as an output gate of this gate.

Parameters **other** (*gate*) – Gate to be designated as an output gate of this gate.

```
>>> c = circuit()
>>> g0 = c.gate(op.id_, is_input=True)
>>> g1 = c.gate(op.id_, is_input=True)
>>> g2 = c.gate(op.and_, [g0, g1])
>>> g3 = c.gate(op.id_, [g2], is_output=True)
```

(continues on next page)

(continued from previous page)

```
>>> g2.output(g3) # Confirm this is idempotent.
>>> c.count()
4
```

class circuit.circuit.**gates**(*iterable=()*, */*)

Bases: list

Data structure for a collection of gates. It is usually assumed that the gates within an instance of this class are related (*e.g.*, they are all part of the same circuit, as is the case when an instance of this class is found as the `gates` attribute of a *circuit* instance) or, at least, interconnected. However, an instance of this class could be used to represent any collection of gates.

static mark(*g*)

Mark all gates reachable from the supplied gate via recursive traversal of input gate references.

Parameters **g** (*gate*) – Gate from which to mark all reachable gates (via input references).

```
>>> c = circuit()
>>> g0 = c.gate(op.id_, is_input=True)
>>> g1 = c.gate(op.id_, is_input=True)
>>> g2 = c.gate(op.and_, [g0, g1])
>>> g3 = c.gate(op.id_, [g2], is_output=True)
>>> gates.mark(g3)
>>> all(g.is_marked for g in [g0, g1, g2, g3])
True
```

gate(*operation=None*, *inputs=None*, *outputs=None*, *is_input=False*, *is_output=False*)

Add a gate with the specified attribute values to this collection of gates.

Parameters

- **operation** (Optional[logical]) – Logical operation that the gate represents.
- **inputs** (Optional[Sequence[Optional[*gate*]]]) – List of input gate object references.
- **outputs** (Optional[Sequence[*gate*]]) – List of output gate object references.
- **is_input** (bool) – Flag indicating if this is an input gate for a circuit.
- **is_output** (bool) – Flag indicating if this is an output gate for a circuit.

The *circuit.gate* method is a wrapper for the *gates.gate* method that belongs that circuit's associated *gates* instance (that instance being stored under the circuit's `gates` attribute).

inputs()

Construct a sequence consisting of all *gate* objects and `None` placeholder entries that appear as inputs to *gate* objects in this instance. For any *gate* instance that does not have any inputs specified, it is automatically treated as if the correct number of inputs (based on the arity of the operation corresponding to that gate) is specified using `None` placeholder entries.

```
>>> gs = gates()
>>> g0 = gs.gate(op.id_, [None])
>>> g1 = gs.gate(op.id_, [None])
>>> g2 = gs.gate(op.not_, [g0])
>>> g3 = gs.gate(op.and_, [g1, g2])
>>> g4 = gs.gate(op.not_, [g3])
>>> g5 = gs.gate(op.not_, [g3])
>>> gates([g0, g1]).inputs()
```

(continues on next page)

(continued from previous page)

```
[None, None]
>>> gates([g0, g1, g2, g5]).inputs() == [None, None, g3]
True
```

Duplicate *gate* entries may appear in the result if the same *gate* object is an input for multiple *gate* objects in this instance.

```
>>> gates([g4, g5]).inputs() == [g3, g3]
True
```

Return type Sequence[Optional[*gate*]]

outputs()

Construct a sequence of gates consisting of all *gate* objects that appear as outputs of *gate* objects in this instance.

```
>>> gs = gates()
>>> g0 = gs.gate(op.id_, [None])
>>> g1 = gs.gate(op.id_, [None])
>>> g2 = gs.gate(op.not_, [g0])
>>> g3 = gs.gate(op.and_, [g1, g2])
>>> g4 = gs.gate(op.not_, [g3])
>>> g5 = gs.gate(op.not_, [g3])
>>> gates([g0, g1]).outputs() == [g2, g3]
True
>>> gates([g4, g5]).outputs()
[]
```

Duplicate *gate* entries may appear in the result if the same *gate* object is an output for multiple *gate* objects in this instance.

```
>>> gates([g0, g1, g2, g5]).outputs() == [g3, g3]
True
```

Return type Sequence[*gate*]

sources()

Construct a gate sequence consisting of all *gate* objects in this instance that have no inputs specified, or have at least one input that is either specified with a placeholder `None` or is a *gate* instance that does not appear in this gate list.

```
>>> gs = gates()
>>> g0 = gs.gate(op.id_, [])
>>> g1 = gs.gate(op.id_, [])
>>> g2 = gs.gate(op.not_, [g0])
>>> g3 = gs.gate(op.and_, [None, g2])
>>> g4 = gs.gate(op.not_, [g3])
>>> g5 = gs.gate(op.not_, [g3])
>>> gates([g0, g1, g2, g3]).sources() == [g0, g1, g3]
True
>>> gates([g0, g2, g4]).sources() == [g0, g4]
True
```

Return type Sequence[*gate*]

sinks()

Construct a gate sequence consisting of all *gate* objects in this instance whose outputs are not consumed by other gates in this instance (though they may have output gates that occur outside of this instance).

```
>>> gs = gates()
>>> g0 = gs.gate(op.id_, [])
>>> g1 = gs.gate(op.id_, [])
>>> g2 = gs.gate(op.not_, [g0])
>>> g3 = gs.gate(op.and_, [g1, g2])
>>> g4 = gs.gate(op.not_, [g3])
>>> g5 = gs.gate(op.not_, [g3])
>>> gates([g0, g1, g2, g3]).sinks() == [g3]
True
>>> gates([g0, g2, g4]).sinks() == [g2, g4]
True
```

Return type Sequence[*gate*]

discard(*g*)

Remove the specified gate from all input and output lists of any *gate* objects in this gate list, and then delete it from this gate list. If the specified gate appears in an input list of another *gate* instance, the placeholder None replaces it.

Parameters **g** (*gate*) – Gate that must be removed from this instance.

```
>>> gs = gates()
>>> g0 = gs.gate(op.id_, [])
>>> g1 = gs.gate(op.not_, [g0])
>>> g2 = gs.gate(op.and_, [g0, g1])
>>> g3 = gs.gate(op.not_, [g2])
>>> gs.discard(g2)
>>> gs.to_legible()
(('id',), ('not', 0), ('not', None))
```

replace(*old*, *new*)

Replace a collection of gates with a different collection of gates, stitching together the new collection of gates with the input and output gates to which the old collection was connected.

Parameters

- **old** (*gates*) – Gate collection that must be removed.
- **new** (*gates*) – Gate collection that must replace the removed gate collection.

As an example, suppose that the gate collection below is constructed.

```
>>> gs = gates()
>>> g0 = gs.gate(op.id_, [])
>>> g1 = gs.gate(op.id_, [])
>>> g2 = gs.gate(op.not_, [g0])
>>> g3 = gs.gate(op.and_, [g1, g2])
>>> g4 = gs.gate(op.not_, [g3])
>>> g5 = gs.gate(op.not_, [g3])
```

It is possible to construct another gate collection `hs` and to replace a portion of `gs` with it. The *gate* instances to be replaced are discarded (using the *discard* method).

```
>>> hs = gates()
>>> h0 = hs.gate(op.xor_, [None, None])
>>> h1 = hs.gate(op.not_, [h0])
>>> gs.replace(gates([g2, g3]), gates(hs))
>>> gs.to_legible()
(('id',), ('id',), ('xor', 0, 1), ('not', 2), ('not', 3), ('not', 3))
```

The replacement occurs in-place and modifies the instance `gs`. Subsequent replacement operations can be performed on the same *gates* instance.

```
>>> js = gates()
>>> j0 = js.gate(op.or_, [None, None])
>>> j1 = js.gate(op.id_, [j0])
>>> gs.replace(gates(hs), gates(js))
>>> gs.to_legible()
(('id',), ('id',), ('or', 0, 1), ('id', 2), ('not', 3), ('not', 3))
```

Note in the example below that if a *gate* instance does not appear as a sink of the gate collection to be replaced, it may not appear as an input in any other gate. In the example below, the `op.not_` gate `h1` is an input to `g5` but is not a sink in the gate collection `ks`.

```
>>> ks = gates()
>>> k0 = ks.gate(op.imp_, [None, None])
>>> k1 = ks.gate(op.id_, [k0])
>>> gs.replace(gates(js + [g4]), gates(ks))
Traceback (most recent call last):
  ...
ValueError: cannot replace a gate that is not a sink ... that collection
```

All gates in the old collection must already be in this instance.

```
>>> gs = gates()
>>> g0 = gs.gate(op.not_, [])
>>> hs = gates()
>>> h0 = hs.gate(op.id_, [])
>>> gs.replace(gates([h0]), gates([h0]))
Traceback (most recent call last):
  ...
ValueError: not all gates to be replaced appear in the gate collection
```

None of the gates in the new collection may appear in this instance before replacement occurs.

```
>>> gs = gates()
>>> g0 = gs.gate(op.not_, [])
>>> gs.replace(gates([g0]), gates([g0]))
Traceback (most recent call last):
  ...
ValueError: one or more replacement gates already appear in the gate collection
```

The gate collection below is used for the remaining examples.

```
>>> gs = gates()
>>> g0 = gs.gate(op.id_, [])
>>> g1 = gs.gate(op.id_, [])
>>> g2 = gs.gate(op.not_, [g0])
>>> g3 = gs.gate(op.and_, [g1, g2])
>>> g4 = gs.gate(op.not_, [g3])
>>> g5 = gs.gate(op.not_, [g3])
```

The gate collection that must be replaced and its replacement must have the same number of inputs and the same number of sink gates.

```
>>> hs = gates()
>>> h0 = hs.gate(op.not_, [None])
>>> gs.replace(gates([g3]), gates(hs))
Traceback (most recent call last):
  ...
ValueError: gate collection to be replaced and its ... same number of inputs
>>> hs = gates()
>>> h0 = hs.gate(op.xor_, [])
>>> h1 = hs.gate(op.id_, [h0])
>>> h2 = hs.gate(op.not_, [h0])
>>> gs.replace(gates([g3]), gates(hs))
Traceback (most recent call last):
  ...
ValueError: gate collection to be replaced and its ... same number of sink gates
```

If a gate collection and its replacement both have the same number of sink gates, then the old and new sink gates are associated with one another according to their position. Each new sink is connected to the output *gate* instance of the corresponding old sink. The example below also illustrates that if a replacement *gate* object has no inputs specified, this method assumes that the appropriate number of placeholder `None` inputs (corresponding to the arity of the operation of that *gate* object) is present.

```
>>> gs.to_legible()
(('id',), ('id',), ('not', 0), ('and', 1, 2), ('not', 3), ('not', 3))
>>> hs = gates()
>>> h0 = hs.gate(op.xor_, []) # Empty input list understood to be ``[None,␣
↪None]``.
>>> h1 = hs.gate(op.id_, [h0])
>>> h2 = hs.gate(op.not_, [h0])
>>> gs.replace(gates([g4, g5]), gates(hs))
>>> gs.to_legible()
(('id',), ('id',), ('not', 0), ('and', 1, 2), ('xor', 3, 3), ('id', 4), ('not',␣
↪4))
```

evaluate(*input*)

Evaluate the collection of gates in this instance, drawing from the supplied input whenever an individual *gate* object either has no specified input gates or has input gates that do not appear in this instance of *gates*.

Parameters **input** (Iterable[int]) – Input bit vector.

This method is provided primarily to enable the evaluation of subsets of gate collections. In the example below, the entire collection of gates in an instance is evaluated on two inputs.

```
>>> gs = gates()
>>> g0 = gs.gate(op.id_, [])
>>> g1 = gs.gate(op.and_, [])
>>> g2 = gs.gate(op.not_, [g0])
>>> g3 = gs.gate(op.xor_, [g1, g2])
>>> g4 = gs.gate(op.not_, [g3])
>>> gs.evaluate([1, 1, 1])
[0]
>>> gs.evaluate([1, 0, 1])
[1]
```

In the example below, a new instance is constructed that contains only a subset of the *gate* instances that are found in the example above. Note that the supplied input is consumed in order to determine the sole argument for the operation of `g2` and the left-hand argument for the operation of `g3`.

```
>>> hs = gates([g2, g3, g4])
>>> hs.evaluate([1, 1])
[0]
>>> [hs.evaluate([x, y]) for x in (0, 1) for y in (0, 1)]
[[0], [1], [1], [0]]
```

Note that this method is *sensitive to the order in which gates appear*, as *gate* objects are evaluated in the order in which they are encountered during an iteration of this instance.

```
>>> gs = gates()
>>> g0 = gs.gate(op.not_, [])
>>> g1 = gs.gate(op.id_, [])
>>> gs.evaluate([0, 1])
[1, 1]
>>> hs = gates([g1, g0])
>>> hs.evaluate([0, 1])
[0, 0]
```

Each *gate* instance must either have no input gates specified, or must have all input gates specified (though it is acceptable for those input gates not to be found in this *gates* instance or even to be specified using the placeholder `None`). This is because, otherwise, there is no way to unambiguously determine which argument(s) may be missing for operations having arities of two or greater.

```
>>> gs = gates()
>>> g0 = gs.gate(op.id_, [])
>>> g1 = gs.gate(op.imp_, [None, g0])
>>> gs.evaluate([0, 1])
[0]
>>> gs = gates()
>>> g0 = gs.gate(op.id_, [])
>>> g1 = gs.gate(op.imp_, [g0, None])
>>> gs.evaluate([0, 1])
[1]
>>> gs = gates()
>>> g0 = gs.gate(op.not_, [])
>>> g1 = gs.gate(op.and_, [g0, None])
>>> del g1.inputs[1]
>>> gs.evaluate([0, 1])
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
  ...
ValueError: number of gate input entries does not match gate operation arity
```

Return type Sequence[int]

to_logical()

Convert an instance into the boolean function to which it corresponds (represented as an instance of the logical.logical.logical class). The running time and memory usage of this method are **exponential in the number of required inputs**.

```
>>> gs = gates()
>>> g0 = gs.gate(op.id_, [])
>>> g1 = gs.gate(op.and_, [])
>>> g2 = gs.gate(op.not_, [g0])
>>> g3 = gs.gate(op.xor_, [g1, g2])
>>> g4 = gs.gate(op.not_, [g3])
>>> gs.to_logical()
(0, 0, 0, 1, 1, 1, 1, 0)
```

Any attempt to convert an instance that has more than one output raises an exception.

```
>>> gs = gates()
>>> g0 = gs.gate(op.id_, [])
>>> g1 = gs.gate(op.and_, [])
>>> gs.to_logical()
Traceback (most recent call last):
  ...
ValueError: gate collection must have exactly one output when evaluated
```

Return type logical

to_immutable()

Return a canonical immutable representation of the list of gates represented by this instance.

```
>>> c = circuit()
>>> g0 = c.gate(op.id_, is_input=True)
>>> g1 = c.gate(op.id_, is_input=True)
>>> g2 = c.gate(op.not_, [g0])
>>> g3 = c.gate(op.not_, [g1])
>>> g4 = c.gate(op.xor_, [g2, g3])
>>> g5 = c.gate(op.id_, [g4], is_output=True)
>>> c.gates.to_immutable()
(((0, 1),), ((0, 1),), ((1, 0), 0), ((1, 0), 1), ((0, 1, 1, 0), 2, 3), ((0, 1),␣
↪4))
```

Immutable objects can be useful for performing comparisons or for using container types such as set.

```
>>> c.gates.to_immutable() == c.gates.to_immutable()
True
>>> len({c.gates.to_immutable(), c.gates.to_immutable()})
1
```

Placeholder gate inputs are permitted if a gate collection is constructed on its own.

```
>>> gs = gates()
>>> g0 = gs.gate(op.id_, [None])
>>> g1 = gs.gate(op.not_, [g0])
>>> gs.to_immutable()
(((0, 1), None), ((1, 0), 0))
```

Return type tuple

`to_legible()`

Return a canonical human-readable representation of the list of gates represented by this instance.

```
>>> c = circuit()
>>> g0 = c.gate(op.id_, is_input=True)
>>> g1 = c.gate(op.id_, is_input=True)
>>> g2 = c.gate(op.not_, [g0])
>>> g3 = c.gate(op.not_, [g1])
>>> g4 = c.gate(op.xor_, [g2, g3])
>>> g5 = c.gate(op.id_, [g4], is_output=True)
>>> c.gates.to_legible()
(('id',), ('id',), ('not', 0), ('not', 1), ('xor', 2, 3), ('id', 4))
```

Placeholder gate inputs are permitted if a gate collection is constructed on its own.

```
>>> gs = gates()
>>> g0 = gs.gate(op.not_, [None])
>>> g1 = gs.gate(op.not_, [g0])
>>> gs.to_legible()
(('not', None), ('not', 0))
```

Return type tuple

class circuit.circuit.**signature**(*input_format=None*, *output_format=None*)

Bases: object

Class for representing circuit signatures (*i.e.*, the input and output bit vector formats associated with evaluation of a circuit).

Parameters

- **input_format** (Optional[Sequence[int]]) – List of bit vector lengths of inputs.
- **output_format** (Optional[Sequence[int]]) – List of bit vector lengths of outputs.

An instance of this class can be used (1) to convert circuit evaluation inputs from the specific signature-compatible format into a flattened list of bits and (2) to convert the flat list of bits obtained as a circuit evaluation output into a signature-compatible format. If a *circuit* instance has been assigned a signature, conversion of inputs and outputs is performed automatically by the `evaluate` method.

```
>>> s = signature([2, 2], [3, 1])
>>> s.input([[1, 0], [0, 1]])
[1, 0, 0, 1]
>>> s.output([1, 1, 0, 0])
[[1, 1, 0], [0]]
```

If no formats are supplied, the signature methods expect that inputs and outputs are flat lists or tuples of integers that represent bits.

```
>>> s = signature()
>>> s.input([1, 0, 1])
[1, 0, 1]
>>> s.input((1, 0, 1))
[1, 0, 1]
>>> s.input([[1], [1], [1]])
Traceback (most recent call last):
  ...
TypeError: input must be a list or tuple of integers
>>> s.input([2, 3, 4])
Traceback (most recent call last):
  ...
ValueError: each bit must be represented by 0 or 1
>>> s.output([1, 0, 1])
[1, 0, 1]
>>> s.output((1, 0, 1))
[1, 0, 1]
```

The conversion methods also perform checks to ensure that the input has valid format, types, and values.

```
>>> s.input({1, 2, 3})
Traceback (most recent call last):
  ...
TypeError: input must be a list or tuple of integers
>>> s = signature([2], [1])
>>> s.input([[2], [3], [4]])
Traceback (most recent call last):
  ...
ValueError: each bit must be represented by 0 or 1
```

Signature specifications must be lists or tuples of integers, where each integer represents the length of an input or output bit vector.

```
>>> signature(['a', 'b'], [1])
Traceback (most recent call last):
  ...
TypeError: signature input format must be a tuple or list of integers
>>> signature([2], ['c'])
Traceback (most recent call last):
  ...
TypeError: signature output format must be a tuple or list of integers
```

input(*input*)

Convert an input organized in a way that matches the signature's input format into a flat list of bits.

Parameters **input** (Sequence[Sequence[int]]) – Input bit vector that matches signature.

```
>>> s = signature(input_format=[2, 3])
>>> s.input([[1, 0], [0, 1, 1]])
[1, 0, 0, 1, 1]
```

Return type Sequence[int]

output(*output*)

Convert a flat list of output bits into a format that matches the signature's output format specification.

Parameters **output** (Sequence[int]) – Flat output bit vector to convert (according to signature).

```
>>> s = signature(output_format=[2, 3])
>>> list(s.output([1, 0, 0, 1, 1]))
[[1, 0], [0, 1, 1]]
```

Return type Sequence[Sequence[int]]

class circuit.circuit.**circuit**(*sig=None*)

Bases: object

Data structure for a circuit instance (with methods that enable counting of gates, pruning of inconsequential gates, and evaluation of the circuit instance on input bit vectors).

Parameters **sig** (Optional[*signature*]) – Signature (input and output bit vector lengths) for the circuit.

Each gate in a circuit is associated with one logical operation. Gate operations are represented using instances of the logical class exported by the logical library. For convenience, the *op* and *operation* constants defined in this module are synonyms for logical.

When programmatically constructing circuits using a *circuit* object's *gate* method, every input and every output must be represented by a dedicated identity gate (for more information on this, see the *gate* method documentation). In the example below, a circuit is constructed that has two input gates, two internal gates, and one output gate.

```
>>> c = circuit()
>>> c.count()
0
>>> g0 = c.gate(op.id_, is_input=True)
>>> g1 = c.gate(op.id_, is_input=True)
>>> g2 = c.gate(op.and_, [g0, g1])
>>> g3 = c.gate(op.or_, [g0, g1]) # Example of gate that can be pruned.
>>> g4 = c.gate(op.id_, [g2], is_output=True)
>>> c.count()
5
```

The gate list associated with a circuit can be converted into a concise human-readable format using the *gates.to_legible* method, enabling manual inspection of the circuit.

```
>>> c.gates.to_legible()
(('id',), ('id',), ('and', 0, 1), ('or', 0, 1), ('id', 2))
```

An instance can be evaluated on any list of bits using the *evaluate* method. The result is a bit vector that includes one bit for each output gate.

```
>>> [list(c.evaluate(bs)) for bs in [[0, 0], [0, 1], [1, 0], [1, 1]]]
[[0], [0], [0], [1]]
```

Using the *prune_and_topological_sort_stable* method, it is possible to remove all internal gates from a circuit from which an output gate cannot be reached. Doing so does not change the order of the input gates or the order of the output gates.

```
>>> c.prune_and_topological_sort_stable()
>>> c.count()
4
>>> [list(c.evaluate(bs)) for bs in [[0, 0], [0, 1], [1, 0], [1, 1]]]
[[0], [0], [0], [1]]
```

It is possible to specify the signature of a circuit using the *signature* class.

```
>>> c = circuit(signature([2], [1]))
>>> c.count()
0
>>> g0 = c.gate(op.id_, is_input=True)
>>> g1 = c.gate(op.id_, is_input=True)
>>> g2 = c.gate(op.not_, [g0])
>>> g3 = c.gate(op.not_, [g1])
>>> g4 = c.gate(op.xor_, [g2, g3])
>>> g5 = c.gate(op.id_, [g4], is_output=True)
>>> c.count()
6
```

Specifying a signature changes the required format for input bit vectors. Rather than a list of integers, the input should consist of a *list* of lists of integers (one list of integers for each input). Thus, an input for the above circuit would be `[[0, 1]]` rather than `[0, 1]` (because the circuit expects one input having two bits). Specifying a signature similarly changes the output format in the same manner, as some circuits may have a signature that indicates that the output consists of some number of bit vectors, each having a specific length. The circuit above has one output: a bit vector having a single bit. Thus, the outputs are of the form `[[1]]`.

```
>>> [list(c.evaluate(bss)) for bss in [[[0, 0]], [[0, 1]], [[1, 0]], [[1, 1]]]]
[[[0]], [[1]], [[1]], [[0]]]
>>> [list(c.evaluate(bss)) for bss in [[[0, 0]], [[0, 1]], [[1, 0]], [[1, 1]]]]
[[[0]], [[1]], [[1]], [[0]]]
```

The circuit in the example below is identical to the one in the example above, but has a different signature. Notice that inputs to the *evaluate* method must have a format that conforms to the circuit's signature. In the example below, the inputs now consist of two bit vectors. Thus, what was above an input of the form `[[0, 1]]` must instead be `[[0], [1]]` (*i.e.*, two inputs each having one bit).

```
>>> c = circuit(signature([1, 1], [1]))
>>> g0 = c.gate(op.id_, is_input=True)
>>> g1 = c.gate(op.id_, is_input=True)
>>> g2 = c.gate(op.not_, [g0])
>>> g3 = c.gate(op.not_, [g1])
>>> g4 = c.gate(op.xor_, [g2, g3])
>>> g5 = c.gate(op.id_, [g4], is_output=True)
>>> [list(c.evaluate(bss)) for bss in [[[0], [0]], [[0], [1]], [[1], [0]], [[1],␣
↪[1]]]]
[[[0]], [[1]], [[1]], [[0]]]
```

The signature of a circuit instance `c` is stored in the attribute `c.signature`. It is possible to update the signature for a circuit by assigning the signature to this attribute. The example below reverts the signature of the circuit `c` defined above to the default (*i.e.*, one input and one output).

```
>>> c.signature = signature()
```

(continues on next page)

(continued from previous page)

```
>>> [list(c.evaluate(bs)) for bs in [[0, 0], [0, 1], [1, 0], [1, 1]]]
[[0], [1], [1], [0]]
```

Circuits can contain constant gates that take no inputs. These correspond to one of the two nullary logical operations that appear in the set `nullary` defined in the logical library). This also implies that circuits that take no inputs can be defined and evaluated.

```
>>> c = circuit()
>>> g0 = c.gate(op.nt_)
>>> g1 = c.gate(op.nf_)
>>> g2 = c.gate(op.or_, [g0, g1])
>>> g3 = c.gate(op.id_, [g2], is_output=True)
>>> c.evaluate([])
[1]
```

A signature can also be used to indicate that a circuit takes no inputs. Note that if a signature is supplied for such a circuit, a list of inputs containing one input that contains no bits must still be supplied in the list of inputs to the *evaluate* method.

```
>>> c = circuit(signature([0], [1]))
>>> g0 = c.gate(op.nt_)
>>> g1 = c.gate(op.nf_)
>>> g2 = c.gate(op.or_, [g0, g1])
>>> g3 = c.gate(op.id_, [g2], is_output=True)
>>> c.evaluate([[]])
[[1]]
```

Circuits may also have input gates or internal gates that have no path to any gate that has been designated as an output gate. Such gates may or may not have outgoing connections to other gates (*i.e.*, they may be *non-sinks* or they may be *sinks*). This implies that circuits that consist of two or more disconnected components are permitted.

```
>>> c = circuit()
>>> g0 = c.gate(op.id_, is_input=True) # Input (non-sink) with no path to output.
>>> g1 = c.gate(op.id_, is_input=True) # Input (sink) with no path to output.
>>> g2 = c.gate(op.not_, [g0]) # Internal gate (non-sink) with no path to output.
>>> g3 = c.gate(op.and_, [g0, g2]) # Internal gate (sink) no path to output.
>>> g4 = c.gate(op.nt_)
>>> g5 = c.gate(op.nf_)
>>> g6 = c.gate(op.or_, [g4, g5])
>>> g7 = c.gate(op.id_, [g6], is_output=True)
```

When evaluating a circuit, the input bit vector must include a bit for every input gate (even if some of those gates have no paths to an output gate).

```
>>> c.evaluate([0, 1])
[1]
```

Pruning a circuit will remove interior gates that have no path to any output gate, but will not remove any input gates (preserving the circuit's signature).

```
>>> c.count()
8
>>> c.prune_and_topological_sort_stable()
```

(continues on next page)

(continued from previous page)

```
>>> c.count()
6
>>> [g.operation.name() for g in c.gates]
['id', 'id', 'nt', 'nf', 'or', 'id']
```

gate(*operation=None*, *inputs=None*, *outputs=None*, *is_input=False*, *is_output=False*)

Add a gate with the specified attribute values to this collection of gates.

Parameters

- **operation** (Optional[logical]) – Logical operation that the gate represents.
- **inputs** (Optional[Sequence[*gate*]]) – List of input gate object references.
- **outputs** (Optional[Sequence[*gate*]]) – List of output gate object references.
- **is_input** (bool) – Flag indicating if this is an input gate for a circuit.
- **is_output** (bool) – Flag indicating if this is an output gate for a circuit.

Gate operations are represented using instances of the `logical` class that is exported by the logical library (note that the *op* and *operation* constants defined in this module are synonyms for `logical`).

```
>>> c = circuit()
>>> g0 = c.gate(op.id_, is_input=True)
>>> g1 = c.gate(op.id_, is_input=True)
>>> g2 = c.gate(op.and_, [g0, g1])
>>> g3 = c.gate(op.id_, [g2], is_output=True)
>>> c.count()
4
>>> len(c.gates)
4
```

This library enforces the convention that **every circuit input and every circuit output must have a dedicated identity gate** (distinct from all internal gates). This is to ensure that the number of inputs (and how they are ordered) and the number of outputs (and how they are ordered) is always well-defined and available to the *evaluate* method (even if there is no *signature* associated with the *circuit* instance). Thus, only a gate corresponding to an identity operation can be designated as an input gate or as an output gate.

```
>>> c = circuit()
>>> g0 = c.gate(op.not_, is_input=True)
Traceback (most recent call last):
  ...
ValueError: input gates must correspond to the identity operation
```

```
>>> g0 = c.gate(op.id_, is_input=True)
>>> g4 = c.gate(op.not_, [g0], is_output=True)
Traceback (most recent call last):
  ...
ValueError: output gates must correspond to the identity operation
```

Once a gate is designated as an output gate, it cannot be an input into another gate.

```
>>> g4 = c.gate(op.not_, [g3])
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
  ...
ValueError: output gates cannot be designated as inputs into other gates
```

This method is a wrapper for the *gates.gate* method of this instance's `gates` attribute.

While `None` can be used as a gate input placeholder when a gate is added to a *gates* instance, this is not permitted when adding a gate to a *circuit* instance.

```
>>> c = circuit()
>>> g0 = c.gate(op.id_, is_input=True)
>>> g1 = c.gate(op.and_, [g0, None])
Traceback (most recent call last):
  ...
ValueError: circuit gate inputs must be explicitly identified gates
```

Furthermore, any non-input gate corresponding to an operation with non-zero arity must specify its inputs and the number of inputs must match the operation arity.

```
>>> c = circuit()
>>> g0 = c.gate(op.nf_) # Nullary false, an operation with zero arity.
>>> g1 = c.gate(op.not_)
Traceback (most recent call last):
  ...
ValueError: non-input circuit gate must have its inputs specified
>>> g1 = c.gate(op.id_, is_input=True)
>>> g2 = c.gate(op.and_, [g1])
Traceback (most recent call last):
  ...
ValueError: number of circuit gate inputs must match arity of gate operation
```

count(*predicate=lambda _: ...*)

Count the number of gates that satisfy the supplied predicate. If no predicate is supplied, the total number of gates in the circuit is returned.

Parameters **predicate** (`Callable`[[*gate*], `bool`]) – Function that distinguishes certain gate objects.

```
>>> c = circuit(signature([2], [1]))
>>> g0 = c.gate(op.id_, is_input=True)
>>> g1 = c.gate(op.id_, is_input=True)
>>> g2 = c.gate(op.not_, [g0])
>>> g3 = c.gate(op.not_, [g1])
>>> g4 = c.gate(op.xor_, [g2, g3])
>>> g5 = c.gate(op.id_, [g4], is_output=True)
>>> c.count(lambda g: g.operation == op.id_)
3
>>> c.count()
6
```

Return type `int`

depth(*predicate=lambda _: ...*)

Calculate the maximum circuit depth. This method assumes the circuit has already been pruned and sorted, and counts all gates by default (including input gates, output gates, identity gates, and gates that correspond

to nullary operations).

Parameters **predicate** (Callable[[*gate*], bool]) – Function that distinguishes certain gate objects.

It is possible to calculate depth with respect to a specific subset of gates, such as the AND-depth (*i.e.*, the maximum number of AND gates that cannot be parallelized. Identity gates are ignored by default).

The example below tests this method on a large unbalanced circuit.

```
>>> c = circuit(signature([2], [1]))
>>> g0 = c.gate(op.id_, is_input=True)
>>> g1 = c.gate(op.id_, is_input=True)
>>> g2 = c.gate(op.not_, [g0])
>>> g3 = c.gate(op.not_, [g1])
>>> gk = g2
>>> for _ in range(1000-2):
...     gk = c.gate(op.and_, [gk, g3])
>>> g4 = c.gate(op.xor_, [g2, gk])
>>> g5 = c.gate(op.id_, [g4], is_output=True)
>>> c.depth()
1002
```

The example below tests a circuit containing only unary gates.

```
>>> c = circuit(signature([1], [1]))
>>> g0 = c.gate(op.id_, is_input=True)
>>> g1 = c.gate(op.not_, [g0])
>>> g2 = c.gate(op.not_, [g1])
>>> g3 = c.gate(op.not_, [g2])
>>> g4 = c.gate(op.not_, [g3])
>>> g5 = c.gate(op.not_, [g4])
>>> g6 = c.gate(op.not_, [g5])
>>> g7 = c.gate(op.not_, [g6])
>>> g8 = c.gate(op.not_, [g7])
>>> g9 = c.gate(op.id_, [g8], is_output=True)
>>> c.depth()
10
```

The example below tests a balanced binary tree circuit (an equivalent of the eight-input XOR gate).

```
>>> c = circuit(signature([8], [1]))
>>> g0 = c.gate(op.id_, is_input=True)
>>> g1 = c.gate(op.id_, is_input=True)
>>> g2 = c.gate(op.id_, is_input=True)
>>> g3 = c.gate(op.id_, is_input=True)
>>> g4 = c.gate(op.id_, is_input=True)
>>> g5 = c.gate(op.id_, is_input=True)
>>> g6 = c.gate(op.id_, is_input=True)
>>> g7 = c.gate(op.id_, is_input=True)
>>> g8 = c.gate(op.xor_, [g0, g1])
>>> g9 = c.gate(op.xor_, [g2, g3])
>>> g10 = c.gate(op.xor_, [g4, g5])
>>> g11 = c.gate(op.xor_, [g6, g7])
>>> g12 = c.gate(op.xor_, [g8, g9])
```

(continues on next page)

(continued from previous page)

```
>>> g13 = c.gate(op.xor_, [g10, g11])
>>> g14 = c.gate(op.xor_, [g12, g13])
>>> g15 = c.gate(op.id_, [g14], is_output=True)
>>> c.depth()
5
>>> c.depth(lambda _g: _g.operation == op.xor_)
3
>>> c.depth(lambda _g: _g.operation == op.and_)
0
```

Return type int

prune_and_topological_sort_stable()

Prune any gates from which an output gate cannot be reached and topologically sort the gates (with input gates all in their original order at the beginning and output gates all in their original order at the end).

```
>>> c = circuit(signature([2], [1]))
>>> g0 = c.gate(op.id_, is_input=True)
>>> g1 = c.gate(op.id_, is_input=True)
>>> g2 = c.gate(op.not_, [g0])
>>> g3 = c.gate(op.not_, [g1])
>>> g4 = c.gate(op.xor_, [g2, g3])
>>> g5 = c.gate(op.id_, [g2], is_output=True)
>>> c.count()
6
>>> c.prune_and_topological_sort_stable()
>>> c.count()
4
>>> [g.operation.name() for g in c.gates]
['id', 'id', 'not', 'id']
```

evaluate(*input*)

Evaluate the circuit on an input organized in a way that matches the circuit signature's input format, and return an output that matches the circuit signature's output format.

Parameters **input** (Union[Sequence[int], Sequence[Sequence[int]]]) – Input bit vector or bit vectors.

```
>>> c = circuit()
>>> g0 = c.gate(op.id_, is_input=True)
>>> g1 = c.gate(op.id_, is_input=True)
>>> g2 = c.gate(op.and_, [g0, g1])
>>> g3 = c.gate(op.id_, [g2], is_output=True)
>>> list(c.evaluate([0, 1]))
[0]
```

It is also possible to evaluate a circuit that has a signature specified. Note that in this case, the inputs and outputs must be lists of lists (to reflect that there are multiple inputs).

```
>>> c = circuit(signature([2], [1]))
>>> g0 = c.gate(op.id_, is_input=True)
>>> g1 = c.gate(op.id_, is_input=True)
>>> g2 = c.gate(op.and_, [g0, g1])
>>> g3 = c.gate(op.id_, [g2], is_output=True)
>>> list(c.evaluate([[0, 1]]))
[[0]]
```

Any attempt to evaluate a circuit on an invalid input raises an exception.

```
>>> c.evaluate([0, 0])
Traceback (most recent call last):
  ...
TypeError: input must be a list or tuple of integer lists
```

If a signature has been specified for the circuit, any attempt to evaluate the circuit on an input that does not conform to the signature raises an exception.

```
>>> c.evaluate([[0, 0, 0]])
Traceback (most recent call last):
  ...
ValueError: input format does not match signature
```

Return type Union[Sequence[int], Sequence[Sequence[int]]]

`to_logical()`

Convert a circuit into the boolean function to which it corresponds (represented as an instance of the logical.logical.logical class). The running time and memory usage of this method are **exponential in the number of input gates**.

```
>>> c = circuit()
>>> g0 = c.gate(op.id_, is_input=True)
>>> g1 = c.gate(op.id_, is_input=True)
>>> g2 = c.gate(op.id_, is_input=True)
>>> g3 = c.gate(op.and_, [g0, g1])
>>> g4 = c.gate(op.xor_, [g2, g3])
>>> g5 = c.gate(op.id_, [g4], is_output=True)
>>> c.to_logical()
(0, 1, 0, 1, 0, 1, 1, 0)
```

This method supports circuits that have a signature specified.

```
>>> c = circuit(signature([3], [1]))
>>> g0 = c.gate(op.id_, is_input=True)
>>> g1 = c.gate(op.id_, is_input=True)
>>> g2 = c.gate(op.id_, is_input=True)
>>> g3 = c.gate(op.and_, [g0, g1])
>>> g4 = c.gate(op.xor_, [g2, g3])
>>> g5 = c.gate(op.id_, [g4], is_output=True)
>>> c.to_logical()
(0, 1, 0, 1, 0, 1, 1, 0)
```

Any attempt to convert a circuit that has more than one output gate raises an exception.

```
>>> c = circuit()
>>> g0 = c.gate(op.id_, is_input=True)
>>> g1 = c.gate(op.not_, [g0])
>>> g2 = c.gate(op.id_, [g0])
>>> g3 = c.gate(op.id_, [g1], is_output=True)
>>> g4 = c.gate(op.id_, [g2], is_output=True)
>>> c.to_logical()
Traceback (most recent call last):
  ...
ValueError: circuit must have exactly one output gate
```

Return type logical

PYTHON MODULE INDEX

C

circuit.circuit, 6

INDEX

C

circuit (*class in circuit.circuit*), 17
circuit.circuit
    module, 6
count() (*circuit.circuit.circuit method*), 21

D

depth() (*circuit.circuit.circuit method*), 21
discard() (*circuit.circuit.gates method*), 10

E

evaluate() (*circuit.circuit.circuit method*), 23
evaluate() (*circuit.circuit.gates method*), 12

G

gate (*class in circuit.circuit*), 7
gate() (*circuit.circuit.circuit method*), 20
gate() (*circuit.circuit.gates method*), 8
gates (*class in circuit.circuit*), 8

I

input() (*circuit.circuit.signature method*), 16
inputs() (*circuit.circuit.gates method*), 8

M

mark() (*circuit.circuit.gates static method*), 8
module
    circuit.circuit, 6

O

op (*in module circuit.circuit*), 7
operation (*in module circuit.circuit*), 7
output() (*circuit.circuit.gate method*), 7
output() (*circuit.circuit.signature method*), 17
outputs() (*circuit.circuit.gates method*), 9

P

prune_and_topological_sort_stable() (*circuit.circuit.circuit method*), 23

R

replace() (*circuit.circuit.gates method*), 10

S

signature (*class in circuit.circuit*), 15
sinks() (*circuit.circuit.gates method*), 10
sources() (*circuit.circuit.gates method*), 9

T

to_immutable() (*circuit.circuit.gates method*), 14
to_legible() (*circuit.circuit.gates method*), 15
to_logical() (*circuit.circuit.circuit method*), 24
to_logical() (*circuit.circuit.gates method*), 14