scadnano

Release 0.19.4

David Doty

Mar 28, 2024

CONTENTS:

CHAPTER

ONE

SCADNANO

The *scadnano* Python module is a library for describing synthetic DNA nanostructures (e.g., DNA origami). To install, type *pip install scadnano* at the command line; more detailed installation instructions and troubleshooting tips are at the GitHub repository.

The scadnano project is developed and maintained by the UC Davis Molecular Computing group. Note that cadnano is a separate project, developed and maintained by the Douglas lab at UCSF.

This module is used to write Python scripts creating files readable by scadnano, a web application useful for displaying and manually editing synthetic DNA nanostructures. The purpose of this module is to help automate some of the task of creating DNA designs, as well as making large-scale changes to them that are easier to describe programmatically than to do by hand in scadnano.

If you find scadnano useful in a scientific project, please cite its associated paper:

> scadnano: A browser-based, scriptable tool for designing DNA nanostructures.
>
> David Doty, Benjamin L Lee, and Tristan Stérin.
>
> DNA 2020: *Proceedings of the 26th International Conference on DNA Computing and Molecular Programming*
>
> [paper | BibTeX]

This document describes the API for the scadnano Python package, see the repository for additional documentation, such as installation instructions. There is separate documentation for the scadnano web interface.

This library uses typing hints from the Python typing library. (https://docs.python.org/3/library/typing.html) Each function and method indicate intended types of the parameters. However, due to Python's design, these types are not enforced at runtime. It is suggested to use a static analysis tool such as that provided by an IDE such as PyCharm (https://www.jetbrains.com/pycharm/) to see warnings when the typing rules are violated. Such warnings probably indicate an erroneous usage.

Most of the classes in this module are Python dataclasses (https://docs.python.org/3/library/dataclasses.html) whose fields show up in the documentation. Their types are listed in parentheses after the name of the class; for example *Color* has `int` fields *Color.r*, *Color.g*, *Color.b*. In general it is safe to read these fields directly, but not to write to them directly. Setter methods (named `set_<fieldname>`) are provided for fields where it makes sense to set it to another value than it had originally. However, due to Python naming conventions for dataclass fields and property setters, it is not straightforward to enforce that the fields cannot be written, so the user must take care not to set them.

scadnano.**default_scadnano_file_extension = 'sc'**

Default filename extension when writing a scadnano file.

class scadnano.**Color**(*r: 'Optional[int]' = None*, *g: 'Optional[int]' = None*, *b: 'Optional[int]' = None*, *hex_string: 'InitVar[str]' = None*)

Parameters

- **r** (*Optional[int]*) –

- **g** (*Optional[int]*) –
- **b** (*Optional[int]*) –
- **hex_string** (*InitVar*) –

r: Optional[int] = None

Red component: 0-255.

Optional if *Color.hex_string* is given.

g: Optional[int] = None

Green component: 0-255.

Optional if *Color.hex_string* is given.

b: Optional[int] = None

Blue component: 0-255.

Optional if *Color.hex_string* is given.

hex_string: InitVar = None

Hex color preceded by # sign, e.g., "#ff0000" is red.

Optional if *Color.r*, *Color.g*, *Color.b* are all given.

class scadnano.**ColorCycler**

Calling `next(color_cycler)` on a ColorCycler named `color_cycler` returns a the next *Color* from a fixed size list, cycling after reaching the end of the list.

To choose new colors, set `color_cycler.colors` to a new list of *Color*'s.

property colors: List[*Color*]

The colors that are cycled through when calling `next()` on some *ColorCycler*.

scadnano.**default_scaffold_color = Color(r=0, g=102, b=204)**

Default color for scaffold strand(s).

scadnano.**default_strand_color = Color(r=0, g=0, b=0)**

Default color for non-scaffold strand(s).

class scadnano.**Grid**(*value*)

Represents default patterns for laying out helices in the side view. Each *Grid* except *Grid.none* has an interpretation of a "grid position", which is a 2D integer coordinate (h, v).

square = 'square'

Square lattice. Increasing h moves right and increasing v moves down. (i.e., "computer screen coordinates" rather than Cartesian coordinates where positive y is up.)

hex = 'hex'

Hexagonal lattice. Uses the *"odd-q horizontal layout"* coordinate system described here: https://www.redblobgames.com/grids/hexagons/. Incrementing v moves down. Incrementing h moves down and to the right if h is even, and moves up and to the right if h is odd.

honeycomb = 'honeycomb'

Honeycomb lattice. This consists of all the hex lattice positions except where honeycomb lattice disallows grid positions (h, v) with v even and h a multiple of 3 or v odd and h = 1 + a multiple of 3.

However, we use the same convention as cadnano for encoding honeycomb coordinates; see *misc/cadnano-format-specs/v2.txt*. That convention is different from simply excluding coordinates from the hex lattice.

none = 'none'

No fixed grid.

scadnano.**DNA_base_wildcard = '?'**

Symbol to insert when a DNA sequence has been assigned to a strand through complementarity, but some regions of the strand are not bound to the strand that was just assigned. Also used in case the DNA sequence assigned to a strand is too short; the sequence is padded with *DNA_base_wildcard* to make its length the same as the length of the strand.

class scadnano.**M13Variant**(*value*)

Variants of M13mp18 viral genome. "Standard" variant is p7249. Other variants are longer.

To create a string with the DNA sequence of one of these variants, call the function *m13()*.

p7249 = 'p7249'

"Standard" variant of M13mp18; 7249 bases long, available from, for example

https://www.tilibit.com/collections/scaffold-dna/products/single-stranded-scaffold-dna-type-p7249

https://www.neb.com/products/n4040-m13mp18-single-stranded-dna

http://www.bayoubiolabs.com/biochemicat/vectors/pUCM13/

p7560 = 'p7560'

Variant of M13mp18 that is 7560 bases long. Available from, for example

https://www.tilibit.com/collections/scaffold-dna/products/single-stranded-scaffold-dna-type-p7560

p8064 = 'p8064'

Variant of M13mp18 that is 8064 bases long. Available from, for example

https://www.tilibit.com/collections/scaffold-dna/products/single-stranded-scaffold-dna-type-p8064

p8634 = 'p8634'

Variant of M13mp18 that is 8634 bases long. At the time of this writing, not listed as available from any biotech vender, but Tilibit will make it for you if you ask. (https://www.tilibit.com/pages/contact-us)

scadnano.**m13**(*rotation=5587*, *variant=M13Variant.p7249*)

The M13mp18 DNA sequence (commonly called simply M13).

By default, starts from cyclic rotation 5587 (with 0-based indexing; commonly this is called rotation 5588, which assumes that indexing begins at 1), as defined in GenBank.

By default, returns the "standard" variant of consisting of 7249 bases, sold by companies such as Tilibit and New England Biolabs.

The actual M13 DNA strand itself is circular, so assigning this sequence to the scaffold *Strand* in a *Design* means that the "5' end" of the scaffold *Strand* (which is a fiction since the actual circular DNA strand has no endpoint) will have the sequence starting at position 5587 (if another value for *rotation* is not specified) starting at the displayed 5' in scadnano, assigned until the displayed 3' end. Assuming the displayed scaffold *Strand* has length $n < 7249$, then a loopout of length $7249 - n$ consisting of the undisplayed bases will be present in the actual DNA structure.

For a more detailed discussion of why this particular rotation of M13 is chosen as the default, see Supplementary Note S8 in [Folding DNA to create nanoscale shapes and patterns. Paul W. K. Rothemund, Nature 440:297-302 (2006)].

Parameters

- **rotation** (*int*) – rotation of circular strand. Valid values are 0 through length-1.
- **variant** (M13Variant) – variant of M13 strand to use

Returns
M13 strand sequence

Return type
str

class scadnano.**ModificationType**(*value*)

Type of modification (5', 3', or internal).

five_prime = "5'"

5' modification type

three_prime = "3'"

3' modification type

internal = 'internal'

internal modification type

class scadnano.**Modification**(*display_text*, *vendor_code*, *connector_length=4*)

Abstract case class of modifications (to DNA sequences, e.g., biotin or Cy3). Use concrete subclasses *Modification3Prime*, *Modification5Prime*, or *ModificationInternal* to instantiate.

Modification.vendor_code is used as a unique ID. Each *Modification.vendor_code* must be unique. This can cause problems with some vendors such as Eurofins (https://eurofinsgenomics.com/en/products/dnarna-synthesis/mods/) that reuse the same vendor code such as [BIOTEG]. See issue https://github.com/UC-Davis-molecular-computing/scadnano-python-package/issues/283.

For example if you create a 5' modification to represent 6 T bases: `t6_5p = Modification5Prime(display_text='6T', vendor_code='TTTTTT')` (this was a useful hack for putting single-stranded extensions on strands before the *Extension* class was created to directly support this idea), then this would clash with a similar 3' modification without specifying unique IDs for them: `t6_3p = Modification3Prime(display_text='6T', vendor_code='TTTTTT') # ERROR`.

In general it is recommended to create a single *Modification* object for each *type* of modification in the design. For example, if many strands have a 5' biotin, then it is recommended to create a single *Modification* object and re-use it on each strand with a 5' biotin:

```
biotin_5p = Modification5Prime(display_text='B', vendor_code='/5Biosg/')
design.draw_strand(0, 0).move(8).with_modification_5p(biotin_5p)
design.draw_strand(1, 0).move(8).with_modification_5p(biotin_5p)
```

Parameters

- **display_text** (*str*) –
- **vendor_code** (*str*) –
- **connector_length** (*int*) –

display_text: str

Short text to display in the web interface as an "icon" visually representing the modification, e.g., `'B'` for biotin or `'Cy3'` for Cy3. This can be arbitrary Unicode; for example, to represent a fluorophore, one can use the "glowing star" symbol , or to represent a quencher, one can use the "large black circle" symbol .

vendor_code: str

Text string specifying this modification used by a vendor (a DNA synthesis company such as IDT). For example, for IDT DNA (https://www.idtdna.com/), use '/5Biosg/' for 5' biotin.

This field must be unique to the *Modification*; undefined behavior could result if two different *Modification* objects have the same *Modification.vendor_code*.

connector_length: int = 4

Length of "connector" displayed in web interface.

Drawn like a carbon chain to offset the display of the modification vertically from the DNA strand. This field is useful for putting two nearby modifications at different heights so that their text does not overlap.

Set the length to 0 to not draw a connector.

class scadnano.**Modification5Prime**(*display_text*, *vendor_code*, *connector_length=4*)

5' modification of DNA sequence, e.g., biotin or Cy3.

In general it is recommended to create a single *Modification* object for each *type* of modification in the design. For example, if many strands have a 5' biotin, then it is recommended to create a single *Modification* object and re-use it on each strand with a 5' biotin:

```
biotin_5p = Modification5Prime(display_text='B', vendor_code='/5Biosg/')
design.draw_strand(0, 0).move(8).with_modification_5p(biotin_5p)
design.draw_strand(1, 0).move(8).with_modification_5p(biotin_5p)
```

Parameters

- **display_text** (*str*) –
- **vendor_code** (*str*) –
- **connector_length** (*int*) –

display_text: str

Short text to display in the web interface as an "icon" visually representing the modification, e.g., `'B'` for biotin or `'Cy3'` for Cy3. This can be arbitrary Unicode; for example, to represent a fluorophore, one can use the "glowing star" symbol , or to represent a quencher, one can use the "large black circle" symbol .

vendor_code: str

Text string specifying this modification used by a vendor (a DNA synthesis company such as IDT). For example, for IDT DNA (https://www.idtdna.com/), use '/5Biosg/' for 5' biotin.

This field must be unique to the *Modification*; undefined behavior could result if two different *Modification* objects have the same *Modification.vendor_code*.

class scadnano.**Modification3Prime**(*display_text*, *vendor_code*, *connector_length=4*)

3' modification of DNA sequence, e.g., biotin or Cy3.

In general it is recommended to create a single *Modification* object for each *type* of modification in the design. For example, if many strands have a 3' biotin, then it is recommended to create a single *Modification* object and re-use it on each strand with a 3' biotin:

```
biotin_3p = Modification3Prime(display_text='B', vendor_code='/3Bio/')
design.draw_strand(0, 0).move(8).with_modification_3p(biotin_3p)
design.draw_strand(1, 0).move(8).with_modification_3p(biotin_3p)
```

Parameters

- **display_text** (*str*) –
- **vendor_code** (*str*) –
- **connector_length** (*int*) –

display_text: str

Short text to display in the web interface as an "icon" visually representing the modification, e.g., `'B'` for biotin or `'Cy3'` for Cy3. This can be arbitrary Unicode; for example, to represent a fluorophore, one can use the "glowing star" symbol , or to represent a quencher, one can use the "large black circle" symbol .

vendor_code: str

Text string specifying this modification used by a vendor (a DNA synthesis company such as IDT). For example, for IDT DNA (https://www.idtdna.com/), use '/5Biosg/' for 5' biotin.

This field must be unique to the *Modification*; undefined behavior could result if two different *Modification* objects have the same *Modification.vendor_code*.

class scadnano.**ModificationInternal**(*display_text*, *vendor_code*, *connector_length=4*, *allowed_bases=None*)

Internal modification of DNA sequence, e.g., biotin or Cy3.

Parameters

- **display_text** (*str*) –
- **vendor_code** (*str*) –
- **connector_length** (*int*) –
- **allowed_bases** (*Optional[AbstractSet[str]]*) –

allowed_bases: Optional[AbstractSet[str]] = None

If None, then this is an internal modification that goes between bases. In this case, the key *Strand.modifications_int* specifying the position of the internal modification is interpreted to mean that the modification goes *after* the base at that position. (For example, this is the parameter *idx* in *StrandBuilder.with_modification_internal()*.)

If instead it is a list of bases, then this is an internal modification that attaches to a base, and this lists the allowed bases for this internal modification to be placed at. For example, internal biotins for IDT must be at a T. If any base is allowed, it should be `{'A','C','G','T'}`.

class scadnano.**Position3D**(*x=0*, *y=0*, *z=0*)

Position (x,y,z) in 3D space.

Parameters

- **x** (*float*) –
- **y** (*float*) –
- **z** (*float*) –

x: float = 0

x-coordinate of position. Increasing *x* moves right in the side view and out of the screen in the main view.

y: float = 0

y-coordinate of position. Increasing *y* moves down in the side and main views, i.e., "screen coordinates". (though this can be rotated to Cartesian coordinates, where y goes up, by selecting "invert y/z axes" in the View menu of scadnano.)

z: float = 0

z-coordinate of position. Increasing *z* moves right in the main view and into the screen in the side view.

class scadnano.**HelixGroup**(*position=Position3D(x=0, y=0, z=0)*, *pitch=0*, *roll=0*, *yaw=0*, *helices_view_order=None*, *grid=Grid.none*)

Represents a set of properties to apply to a specific group of *Helix*'s in the *Design*.

A *HelixGroup* is useful for grouping together helices that should all be in parallel, as part of a design where different groups are not parallel. In particular, each *HelixGroup* can be given its own 3D position and pitch/yaw/roll orientation angles. Each *HelixGroup* does not actually *contain* its helices; they are associated through the field *Helix.group*, which is a string representing a key in the dict `groups` specified in the constructor for *Design*.

If there are *HelixGroup*'s explicitly specified, then the field *Design.grid* is ignored. Each *HelixGroup* has its own grid, and the fields *Helix.position* or *Helix.grid_position* are considered relative to the origin of that *HelixGroup* (i.e., the value *HelixGroup.position*). Although an individual *Helix* can have a non-zero *Helix.roll* (which is in addition to whatever value there is for *HelixGroup.roll*), all helices in a group are parallel.

The three angles are interpreted to be applied in the following order: first yaw, then pitch, then roll, using the "intrinsic rotation" convention (see https://en.wikipedia.org/wiki/Euler_angles#Conventions_by_intrinsic_rotations). This convention is not apparent in the scadnano web interface, which only directly shows pitch, but it shows up, for example, in oxDNA export via *Design.to_oxdna_format()*. See the fields *HelixGroup.pitch*, *HelixGroup.roll*, and *HelixGroup.yaw* for an explanation how to interpret each rotation.

Parameters

- **position** (Position3D) –
- **pitch** (*float*) –
- **roll** (*float*) –
- **yaw** (*float*) –
- **helices_view_order** (*Optional[List[int]]*) –
- **grid** (Grid) –

position: *Position3D* = Position3D(x=0, y=0, z=0)

The "origin" of this *HelixGroup*.

pitch: float = 0

Angle in the main view plane; 0 means pointing to the right (min_offset on left, max_offset on right).

Rotation is *clockwise* in the main view, i.e., clockwise in the Y-Z plane, around the X-axis, when Y-axis points down, Z-axis points right, and X-axis points out of the page. See https://en.wikipedia.org/wiki/Aircraft_principal_axes. Units are degrees.

roll: float = 0

Same meaning as *Helix.roll*, applied to every *Helix* in the group, i.e., it represents the rotation about the axis of a helix.

Rotation is *clockwise* in the side view, i.e., in the X-Y plane, around the Z-axis, when X-axis points right, Y-axis points down, and Z-axis points into the page.

yaw: float = 0

Third angle for orientation besides *HelixGroup.pitch* and *HelixGroup.roll*. Not visually displayed in scadnano, but here to support more general 3D applications.

Rotation is *clockwise* while looking down onto the main view, i.e., in the X-Z plane, around the Y-axis, when X-axis points down, Z-axis points right, and Y-axis points into the page. See https://en.wikipedia.org/wiki/Aircraft_principal_axes. Units are degrees.

helices_view_order: Optional[List[int]] = None

Order in which to display the *Helix*'s in the group in the 2D view; if None, then the order is given by the order of the fields *Helix.idx* for each *Helix* in this *HelixGroup*.

grid: *Grid* = 'none'

Grid of this *HelixGroup* used to interpret the field *Helix.grid_position*.

helices_view_order_inverse(*idx*)

Given a *Helix.idx* in this *HelixGroup*, return its view order.

Parameters
idx (*int*) – index of *Helix* in this *HelixGroup*

Returns
view order of the *Helix*

Raises
ValueError – if *idx* is not the index of a *Helix* in this *HelixGroup*

Return type
int

class scadnano.**Geometry**(*rise_per_base_pair=0.332*, *helix_radius=1.0*, *bases_per_turn=10.5*, *minor_groove_angle=150.0*, *inter_helix_gap=1.0*)

Parameters controlling some geometric visualization/physical aspects of a *Design*.

Parameters

- **rise_per_base_pair** (*float*) –
- **helix_radius** (*float*) –
- **bases_per_turn** (*float*) –
- **minor_groove_angle** (*float*) –
- **inter_helix_gap** (*float*) –

rise_per_base_pair: float = 0.332

Distance in nanometers between two adjacent base pairs along the length of a DNA double helix.

helix_radius: float = 1.0

Radius of a DNA helix in nanometers.

bases_per_turn: float = 10.5

Number of DNA base pairs in a full turn of DNA.

minor_groove_angle: float = 150.0

Minor groove angle in degrees.

inter_helix_gap: float = 1.0

Gap between helices in nanometers due to electrostatic repulsion. This is used by the scadnano web interface to display an appropriate aspect ratio for 2D DNA structures.

The default value of 1.0 nm is approximately the average distance, as measured by atomic force microscopy (AFM) images, for 2D DNA origami using the *Grid.square* grid, with 32 base pairs in between consecutive crossovers between two helices. Such a structure with *n* parallel helices generally is measured to be about 3`n` nm high on AFM images. Since each DNA helix is 2 nm diameter, this implies an average interhelix gap of 1.0 nm, though of course it is just an average, and the actual gap varies depending on distance to the nearest crossover: at a crossover the distance is close to 0 and halfway between two crossovers, the distance is greater than 1 nm.

This value may be inappropriate for designs with different crossover spacing, for example single-stranded tiles with 21 base pairs between consecutive crossovers. (In that case 0.5 nm seems to be a more appropriate approximation.)

class scadnano.**Helix**(*max_offset=None*, *min_offset=0*, *major_tick_start=None*, *major_tick_distance=None*, *major_tick_periodic_distances=None*, *major_ticks=None*, *grid_position=None*, *position=None*, *roll=0*, *idx=None*, *group='default_group'*, *_domains=<factory>*)

Represents a "helix" where *Domain*'s could go. Technically a *Helix* can contain no *Domain*'s. More commonly, some partial regions of it may have only 1 or 0 *Domain*'s. So it is best thought of as a "potential" double-helix.

It has a 1-dimensional integer coordinate system given by "offsets", integers between *Helix.min_offset* (inclusive) and *Helix.max_offset* (exclusive). At any valid offset for this *Helix*, at most two *Domain*'s may share that offset on this *Helix*, and if there are exactly two, then one must have *Domain.forward* = `true` and the other must have *Domain.forward* = `false`.

Each *Helix* has an index, accessible via *Helix.idx*. By default this is its order in the list of all *Helix*'s (this is how the *Design* constructor sets the field if it is not already set), but it can be manually assigned to be any integer that is unique to the *Helix*. This index is how a *Domain* is associated to the *Helix* via the field *Domain.helix*.

Parameters

- **max_offset** (*Optional[int]*) –
- **min_offset** (*int*) –
- **major_tick_start** (*Optional[int]*) –
- **major_tick_distance** (*Optional[int]*) –
- **major_tick_periodic_distances** (*Optional[List[int]]*) –
- **major_ticks** (*Optional[List[int]]*) –
- **grid_position** (*Optional[Tuple[int, int]]*) –
- **position** (*Optional[Position3D]*) –
- **roll** (*float*) –
- **idx** (*Optional[int]*) –
- **group** (*str*) –
- **_domains** (*List[Domain]*) –

max_offset: Optional[int] = None

Maximum offset (exclusive) of *Domain* that can be drawn on this *Helix*.

Optional field. If unspecified, it is calculated when the *Design* is instantiated as the largest *Domain.end* offset of any *Domain* in the design.

min_offset: int = 0

Minimum offset (inclusive) of *Domain* that can be drawn on this *Helix*.

Optional field. Default value 0.

major_tick_start: Optional[int] = None

Offset of first major tick when not specifying *Helix.major_ticks*. Used in combination with either *Helix.major_tick_distance* or *Helix.major_tick_periodic_distances*.

Optional field. If not specified, is initialized to value *Helix.min_offset*.

major_tick_distance: Optional[int] = None

Distance between major ticks (bold) delimiting boundaries between bases. Major ticks will appear in the visual interface at positions

Optional field. If 0 then no major ticks are drawn. If not specified then the default value is assumed. If the grid is *Grid.square* then the default value is 8. If the grid is *Grid.hex* or *Grid.honeycomb* then the default value is 7.

major_tick_periodic_distances: Optional[List[int]] = None

Periodic distances between major ticks. For example, setting *Helix.major_tick_periodic_distances* = [2, 3] and *Helix.major_tick_start* = 10 means that major ticks will appear at 12, 15, 17, 20, 22, 25, 27, 30, ...

Optional field. *Helix.major_tick_distance* is equivalent to the setting *Helix.major_tick_periodic_distances* = [*Helix.major_tick_distance*].

major_ticks: Optional[List[int]] = None

If not None, overrides *Helix.major_tick_distance* to specify a list of offsets at which to put major ticks.

grid_position: Optional[Tuple[int, int]] = None

(h,v) position of this helix in the side view grid, if *Grid.square*, *Grid.hex* , or *Grid.honeycomb* is used in the *Design* containing this helix. *h* and *v* are in units of "helices": incrementing *h* moves right one helix in the grid and incrementing *v* moves down one helix in the grid. In the case of the hexagonal lattice, The convention is that incrementing *v* moves down and to the right if h is even, and moves down and to the left if *h* is odd. This is the "odd-q" coordinate system here: https://www.redblobgames.com/grids/hexagons/) However, the default y position in the main view for helices does not otherwise depend on grid_position. The default is to list the y-coordinates in order by helix idx.

Default is *h* = 0, *v* = index of *Helix* in *Design.helices*.

In the case of the honeycomb lattice, we use the same convention as cadnano for encoding hex coordinates, see *misc/cadnano-format-specs/v2.txt*. That convention is different from simply excluding coordinates from the hex lattice.

position: Optional[*Position3D*] = None

Position (x,y,z) of this *Helix* in 3D space.

Must be None if *Helix.grid_position* is specified.

roll: float = 0

Angle around the center of the helix; 0 means pointing straight up in the side view.

Rotation is clockwise in the side view; the same convention as *HelixGroup.roll*. Units are degrees.

idx: Optional[int] = None

Index of this *Helix*.

Optional if no other *Helix* specifies a value for *idx*. Default is the order of the *Helix* is listed in constructor for *Design*.

group: str = 'default_group'

Name of the *HelixGroup* to which this *Helix* belongs.

calculate_major_ticks(*grid*)

Calculates full list of major tick marks, whether using *default_major_tick_distance* (from *Design*), *Helix.major_tick_distance*, or *Helix.major_ticks*. They are used in reverse order to determine precedence. (e.g., *Helix.major_ticks* overrides *Helix.major_tick_distance*, which overrides *default_major_tick_distance* from *Design*.

Parameters
grid (Grid) –

Return type
List[int]

calculate_position(*grid*, *geometry=None*)

Parameters

- **grid** (Grid) – *Grid* of this *Helix* used to interpret the field *Helix.grid_position*. Must be None if *Helix.grid_position* is None.
- **geometry** (*Optional[Geometry]*) – *Geometry* parameters to determine distance between helices in a grid. Must be None if *Helix.grid_position* is None.

Returns
Position of this *Helix* in 3D space, based on its *Helix.grid_position* if it is not None, or its *Helix.position* otherwise.

Return type
Position3D

property domains: List[*Domain*]
Return *Domain*'s on this *Helix*. Assigned when a *Design* is created using this *Helix*.

Returns
Domain's on this helix

backbone_angle_at_offset(*offset*, *forward*, *geometry*)
Computes the backbone angle at *offset* for the strand in the direction given by *forward*.

Parameters

- **offset** (*int*) – offset on this helix
- **forward** (*bool*) – whether to compute angle for the forward or reverse strand
- **geometry** (Geometry) – *Geometry* parameters to determine bases per turn

Returns
backbone angle at *offset* for the strand in the direction given by *forward*.

Return type
float

crossover_addresses(*helices*, *allow_intrahelix=True*, *allow_intergroup=True*)

Parameters

- **helices** (*Dict[int, Helix]*) – The dict of helices in which this *Helix* is contained, that contains other helices to which it might be connected by crossovers.
- **allow_intrahelix** (*bool*) – if `False`, then do not return crossovers to the same *Helix* as this *Helix*
- **allow_intergroup** (*bool*) – if `False`, then do not return crossovers to a *Helix* in a different helix group as this *Helix*

Returns
list of triples (*helix_idx*, *offset*, *forward*) of all crossovers incident to this *Helix*, where *offset* is the offset of the crossover and *helix_idx* is the *Helix.idx* of the other *Helix* incident to the crossover.

Return type
List[*Tuple*[int, int, bool]]

relax_roll(*helices*, *grid*, *geometry*)

Like *Design.relax_helix_rolls()*, but only for this *Helix*.

Parameters

- **helices** (*Dict[int, Helix]*) –
- **grid** (Grid) –
- **geometry** (Geometry) –

Return type
None

compute_relaxed_roll_delta(*helices*, *grid*, *geometry*)

Like *Helix.relax_roll()*, but just returns the amount by which to rotate the current roll, without actually altering the field *Helix.roll*.

Parameters

- **helices** (*Dict[int, Helix]*) –
- **grid** (Grid) –
- **geometry** (Geometry) –

Return type
float

scadnano.**angle_from_helix_to_helix**(*helix*, *other_helix*, *grid=None*, *geometry=None*)

Computes angle between *helix* and *other_helix* in degrees.

Parameters

- **helix** (Helix) – first helix
- **other_helix** (Helix) – second helix
- **grid** (*Optional[Grid]*) – *Grid* to use when calculating Helix positions
- **geometry** (*Optional[Geometry]*) – *Geometry* to use when calculating Helix positions

Returns
angle between *helix* and *other_helix* in degrees.

Return type
float

scadnano.**minimum_strain_angle**(*relative_angles*)

Computes the angle that minimizes the "strain" of all relative angles in the given list.

A "relative angle" is a pair (θ, μ). The strain is set to 0 by setting $\theta = \mu$; more generally the strain is $(\theta - \mu)^2$, where $\theta - \mu$ is the "angular difference" (e.g., 10-350 is 20 since 350 is also -10 mod 360).

The constraint is that in the list $[(\theta_1, \mu_1), (\theta_2, \mu_2), \ldots, (\theta_n, \mu_n)]$, we can rotate all angles θ_i by the same amount θ. So this calculates the angle θ that minimizes $\sum_i [(\theta + \theta_i) - \mu_i]^2$

Parameters
relative_angles (*List[Tuple[float, float]]*) – List of (θ_i, μ_i) pairs, where $\theta_i = \mu_i$ means 0 strain, and angles are in units of degrees.

Returns
angle θ by which to rotate all angles θ_i (but not changing any "zero angle" μ_i) such that $\sum_i [(\theta + \theta_i) - \mu_i]^2$ is minimized.

Return type
float

scadnano.**angle_distance**(*x*, *y*)

Parameters

- **x** (*float*) – angle in degrees
- **y** (*float*) – angle in degrees

Returns
signed difference between angles *x* and *y*, in degrees, in range [-180, 180]

Return type
float

scadnano.**sum_squared_angle_distances**(*angles*, *angle*)

Parameters

- **angles** (*List[float]*) – list of angles in degrees
- **angle** (*float*) – angle in degrees

Returns
sum of squared distances from each angle in *angles* to *angle*

Return type
float

scadnano.**average_angle**(*angles*)

Calculate the "circular mean" of the angles in *angles*. Note this coincides with the arithemtic mean for certain lists of angles, e.g., [0, 10, 50], in a way that the circular mean calculated via interpreting angles as unit vectors (https://en.wikipedia.org/wiki/Circular_mean) does not.

This algorithm is due to Julian Panetta. (https://julianpanetta.com/)

Parameters
angles (*List[float]*) – List of angles in degrees.

Returns
average angle of the list of angles, normalized to be between 0 and 360.

Return type
float

class scadnano.**Domain**(*helix*, *forward*, *start*, *end*, *deletions=<factory>*, *insertions=<factory>*, *name=None*, *label=None*, *dna_sequence=None*, *color=None*)

A maximal portion of a *Strand* that is continguous on a single *Helix*. A *Strand* contains a list of *Domain*'s (and also potentially *Loopout*'s).

Parameters

- **helix** (*int*) –
- **forward** (*bool*) –
- **start** (*int*) –
- **end** (*int*) –
- **deletions** (*List[int]*) –
- **insertions** (*List[Tuple[int, int]]*) –

- **name** (*Optional[str]*) –
- **label** (*Optional[str]*) –
- **dna_sequence** (*Optional[str]*) –
- **color** (*Optional[Color]*) –

helix: int

index of the *Helix* on which this *Domain* resides.

forward: bool

Whether the strand "points" forward (i.e., its 3' end has a larger offset than its 5' end). If *Domain.forward* is `True`, then *Domain.start* is the 5' end of the *Domain* and *Domain.end* is the 3' end of the *Domain*. If *Domain.forward* is `False`, these roles are reversed.

start: int

The smallest offset position of any base on this Domain (3' end if *Domain.forward* = `False`, 5' end if *Domain.forward* = `True`).

end: int

1 plus the largest offset position of any base on this Domain (5' end if *Domain.forward* = `False`, 3' end if *Domain.forward* = `True`). Note that the set of base offsets occupied by this Domain is {start, start+1, ..., end-1}, i.e., inclusive for `Strand.start` but exclusive for `Strand.end`, the same convention used in Python for slices of lists and strings. (e.g., `"abcdef"[1:3] == "bc"`)

Some methods (such as *Domain.dna_sequence_in()*) use the convention of being inclusive on both ends and are marked with the word "INCLUSIVE". (Such a convention is easier to reason about when there are insertions and deletions.)

deletions: List[int]

List of positions of deletions on this Domain.

insertions: List[Tuple[int, int]]

List of (position,num_insertions) pairs on this Domain.

This is the number of *extra* bases in addition to the base already at this position. The total number of bases at this offset is num_insertions+1.

name: Optional[str] = None

Optional name to give this *Domain*.

This is used to interoperate with the dsd DNA sequence design package.

label: Optional[str] = None

This can be used to attach a "label" to associate to this *Loopout*.

See *Strand.label* for examples.

dna_sequence: Optional[str] = None

DNA sequence of this Domain, or `None` if no DNA sequence has been assigned to this *Domain*'s *Strand*.

color: Optional[*Color*] = None

Color to show this domain in the main view. If specified, overrides the field *Strand.color*.

strand()

Returns
The *Strand* that contains this *Domain*.

Return type
Strand

vendor_dna_sequence()

Returns
vendor DNA sequence of this *Domain*, or None if no DNA sequence has been assigned. The difference between this and the field *Domain.dna_sequence* is that this will add internal modification codes.

Return type
Optional[str]

set_name(*name*)

Sets name of this *Domain*.

Parameters
name (*str*) –

Return type
None

set_label(*label*)

Sets label of this *Domain*.

Parameters
label (*str*) –

Return type
None

offset_5p()

5' offset of this *Domain*, INCLUSIVE.

Return type
int

offset_3p()

3' offset of this *Domain*, INCLUSIVE.

Return type
int

contains_offset(*offset*)

Indicates if *offset* is the offset of a base on this *Domain*.

Note that offsets refer to visual portions of the displayed grid for the Helix. If for example, this Domain starts at position 0 and ends at 10, and it has 5 deletions, then it contains the offset 7 even though there is no base 7 positions from the start.

Parameters
offset (*int*) –

Return type
bool

dna_length()

Number of bases in this Domain.

Return type
int

dna_length_in(*left*, *right*)

Number of bases in this Domain between offsets *left* and *right* (INCLUSIVE).

Parameters

- **left** (*int*) –
- **right** (*int*) –

Return type
int

`visual_length`()

Distance between *Domain.start* offset and *Domain.end* offset.

This can be more or less than the *Domain.dna_length()* due to insertions and deletions.

Return type
int

`dna_sequence_in`(*offset_left*, *offset_right*)

Return DNA sequence of this Domain in the interval of offsets given by [*offset_left*, *offset_right*], INCLUSIVE, or `None` if no DNA sequence has been assigned to this *Domain*'s *Strand*.

WARNING: This is inclusive on both ends, unlike other parts of this API where the right endpoint is exclusive. This is to make the notion well-defined when one of the endpoints is on an offset with a deletion or insertion.

Parameters

- **offset_left** (*int*) –
- **offset_right** (*int*) –

Return type
Optional[str]

`get_seq_start_idx`()

Starting DNA subsequence index for first base of this *Domain* on its Parent *Strand*'s DNA sequence.

Return type
int

`domain_offset_to_strand_dna_idx`(*offset*, *offset_closer_to_5p*)

Convert from offset on this *Domain*'s *Helix* to string index on the parent *Strand*'s DNA sequence.

If *offset_closer_to_5p* is `True`, (this only matters if *offset* contains an insertion) then the only leftmost string index corresponding to this offset is included, otherwise up to the rightmost string index (including all insertions) is included.

Parameters

- **offset** (*int*) –
- **offset_closer_to_5p** (*bool*) –

Return type
int

`overlaps`(*other*)

Indicates if this *Domain*'s set of offsets (the set $\{x \in \mathbb{N} \mid \texttt{self.start} \leq x \leq \texttt{self.end}\}$) has nonempty intersection with those of *other*, and they appear on the same helix, and they point in opposite directions.

Parameters
other (Domain) –

Return type
bool

`overlaps_illegally`(*other*)

Indicates if this *Domain*'s set of offsets (the set $\{x \in \mathbb{N} \mid \texttt{self.start} \leq x \leq \texttt{self.end}\}$) has nonempty intersection with those of *other*, and they appear on the same helix, and they point in the same direction.

Parameters
other (Domain) –

Return type
bool

`compute_overlap`(*other*)

Return [left,right) offset indicating overlap between this Domain and *other*.

Return `(-1,-1)` if they do not overlap (different helices, or non-overlapping regions of the same helix).

Parameters
other (Domain) –

Return type
Tuple[int, int]

`insertion_offsets`()

Return offsets of insertions (but not their lengths).

Return type
List[int]

`is_extreme_domain`(*five_prime*)

Parameters
five_prime (*bool*) – whether to ask about 5' end or 3' end

Returns
Whether this *Domain* is the 5' or 3' most *Domain* on its *Strand*. (which depends on parameter *five_prime*

Return type
bool

`is_5p_domain`()

Returns
Whether this *Domain* is the 5' most *Domain* on its *Strand*.

Return type
bool

`is_3p_domain`()

Returns
Whether this *Domain* is the 3' most *Domain* on its *Strand*.

Return type
bool

`class scadnano.Loopout`(*length*, *name=None*, *label=None*, *dna_sequence=None*, *color=None*)

Represents a single-stranded loopout on a *Strand*.

One could think of a *Loopout* as a type of *Domain*, but none of the fields of *Domain* make sense for *Loopout*, so they are not related to each other in the type hierarchy. It is interpreted that a *Loopout* is a single-stranded

region bridging two *Domain*'s that are connected to *Helix*'s. It is illegal for two consecutive *Domain*'s to both be *Loopout*'s, or for a *Loopout* to occur on either end of the *Strand* (i.e., each *Strand* must begin and end with a *Domain* or *Extension*).

For example, one use of a loopout is to describe a hairpin (a.k.a., stem-loop). The following creates a *Strand* that represents a hairpin with a stem length of 10 and a loop length of 5.

```
import scadnano as sc

domain_f = sc.Domain(helix=0, forward=True, start=0, end=10)
loop = sc.Loopout(length=5)
domain_r = sc.Domain(helix=0, forward=False, start=0, end=10)
hairpin = sc.Strand([domain_f, loop, domain_r])
```

It can also be created with chained method calls

```
import scadnano as sc

design = sc.Design(helices=[sc.Helix(max_offset=10)])
design.draw_strand(0,0).move(10).loopout(0,5).move(-10)
```

Parameters

- **length** (*int*) –
- **name** (*Optional[str]*) –
- **label** (*Optional[str]*) –
- **dna_sequence** (*Optional[str]*) –
- **color** (*Optional[Color]*) –

length: int

Length (in DNA bases) of this *Loopout*.

name: Optional[str] = None

Optional name to give this *Loopout*.

This is used to interoperate with the dsd DNA sequence design package.

label: Optional[str] = None

This can be used to attach a "label" to associate to this *Loopout*.

See *Strand.label* for examples.

dna_sequence: Optional[str] = None

DNA sequence of this *Loopout*, or `None` if no DNA sequence has been assigned.

color: Optional[*Color*] = None

Color to show this loopout in the main view. If specified, overrides the field *Strand.color*.

strand()

Returns

The *Strand* that contains this *Loopout*.

Return type

Strand

vendor_dna_sequence()

Returns
vendor DNA sequence of this *Loopout*, or None if no DNA sequence has been assigned. The difference between this and the field *Loopout.dna_sequence* is that this will add internal modification codes.

Return type
Optional[str]

set_name(*name*)

Sets name of this *Loopout*.

Parameters
name (*str*) –

Return type
None

set_label(*label*)

Sets label of this *Loopout*.

Parameters
label (*Optional[str]*) –

Return type
None

dna_length()

Length of this *Loopout*; same as field *Loopout.length*.

Return type
int

get_seq_start_idx()

Starting DNA subsequence index for first base of this *Loopout* on its *Strand*'s DNA sequence.

Return type
int

class scadnano.**Extension**(*num_bases*, *display_length=1.0*, *display_angle=35.0*, *label=None*, *name=None*, *dna_sequence=None*, *color=None*)

Represents a single-stranded extension on either the 3' or 5' end of *Strand*.

One could think of an *Extension* as a type of *Domain*, but none of the fields of *Domain* make sense for *Extension*, so they are not related to each other in the type hierarchy. It is interpreted that an *Extension* is a single-stranded region that resides on either the 3' or 5' end of the *Strand*. It is illegal for an *Extension* to be placed in the middle of the *Strand* or for an *Extension* to be adjacent to a *Loopout*.

```
import scadnano as sc

domain = sc.Domain(helix=0, forward=True, start=0, end=10)
left_toehold = sc.Extension(num_bases=3)
right_toehold = sc.Extension(num_bases=2)
strand = sc.Strand([left_toehold, domain, right_toehold])
```

It can also be created with chained method calls

```
import scadnano as sc

design = sc.Design(helices=[sc.Helix(max_offset=10)])
design.draw_strand(0,0).extension_5p(3).move(10).extension_3p(2)
```

which makes this strand with *Extension*'s on each side of the length-10 *Domain*:

```
[
 \               >
  \             /
   \           /
    ----------
```

Parameters

- **num_bases** (*int*) –
- **display_length** (*float*) –
- **display_angle** (*float*) –
- **label** (*Optional[str]*) –
- **name** (*Optional[str]*) –
- **dna_sequence** (*Optional[str]*) –
- **color** (*Optional[Color]*) –

num_bases: int

Length (in DNA bases) of this *Extension*.

display_length: float = 1.0

Length (in nm) to display the line representing the *Extension* in the scadnano web app.

display_angle: float = 35.0

Angle (in degrees) to display in the scadnano web app.

This angle is relative to the "rotation frame" of the adjacent domain. 0 degrees means parallel to the adjacent domain. 90 degrees means pointing away from the helix. 180 degrees means means antiparallel to the adjacent domain (overlapping). If a forward strand, will go above the strand; if a reverse strand, will go below, for degrees strictly between 0 and 180.

label: Optional[str] = None

This can be used to attach a "label" to associate to this *Extension*.

See *Strand.label* for examples.

name: Optional[str] = None

Optional name to give this *Extension*.

dna_sequence: Optional[str] = None

DNA sequence of this *Extension*, or `None` if no DNA sequence has been assigned.

color: Optional[*Color*] = None

Color to show this extension in the main view. If specified, overrides the field *Strand.color*.

dna_length()

Length of this *Extension*; same as field *Extension.num_bases*.

Return type
int

strand()

Returns
The *Strand* that contains this *Extension*.

Return type
Strand

vendor_dna_sequence()

Returns
vendor DNA sequence of this *Extension*, or `None` if no DNA sequence has been assigned. The difference between this and the field *Extension.dna_sequence* is that this will add internal modification codes.

Return type
Optional[str]

set_label(*label*)

Sets label of this *Extension*.

Parameters
label (*Optional[str]*) –

Return type
None

set_name(*name*)

Sets name of this *Extension*.

Parameters
name (*str*) –

Return type
None

scadnano.**wc**(*seq*)

Return reverse Watson-Crick complement of *seq*. For example, `wc('AACCTG')` returns `'CAGGTT'`.

Parameters
seq (*str*) – a DNA sequence

Returns
reverse Watson-Crick complement of *seq*.

Return type
str

class scadnano.**VendorFields**(*scale='25nm'*, *purification='STD'*, *plate=None*, *well=None*)

Data required when ordering DNA strands from a synthesis company. These fields were originally designed for IDT (Integrated DNA Technologies) and the default values for *VendorFields.scale* and *VendorFields.purification* reflect that. However, most vendors have the same concepts of scale, purification, a code to specify the modification (the field `VendorFields.vendor_code`), etc., so we use this generic class for any of them. Currently only IDT is supported by methods to automatically export DNA sequences in the format IDT recognizes, but one should be able to write custom code to export other formats that reads the fields in this object.

When exporting to IDT files via *Design.write_idt_plate_excel_file()* or *Design.write_idt_bulk_input_file()*, the field *Strand.name* is used for the name if it exists, otherwise a reasonable default is chosen.

Parameters

- **scale** (*str*) –
- **purification** (*str*) –
- **plate** (*Optional[str]*) –
- **well** (*Optional[str]*) –

scale: str = '25nm'

Synthesis scale at which to synthesize the strand (third field in IDT bulk input: https://www.idtdna.com/site/order/oligoentry). Choices supplied by IDT at the time this was written: `"25nm"`, `"100nm"`, `"250nm"`, `"1um"`, `"5um"`, `"10um"`, `"4nmU"`, `"20nmU"`, `"PU"`, `"25nmS"`.

purification: str = 'STD'

Purification options (fourth field in IDT bulk input: https://www.idtdna.com/site/order/oligoentry). Choices supplied by IDT at the time this was written: `"STD"`, `"PAGE"`, `"HPLC"`, `"IEHPLC"`, `"RNASE"`, `"DUALHPLC"`, `"PAGEHPLC"`.

plate: Optional[str] = None

Name of plate in case this strand will be ordered on a 96-well or 384-well plate.

Optional field, but non-optional if *VendorFields.well* is not `None`.

well: Optional[str] = None

Well position on plate in case this strand will be ordered on a 96-well or 384-well plate. Well position on plate in case this strand will be ordered on a 96-well or 384-well plate.

Optional field, but non-optional if *VendorFields.plate* is not `None`.

class scadnano.**StrandBuilder**(*design*, *helix*, *offset*)

Represents a *Strand* that is being built in an existing *Design*.

This is an intermediate object created when using chained method building by calling *Design.draw_strand()*, for example

```
design.draw_strand(0, 0).to(10).cross(1).to(5).with_modification_5p(mod.biotin_5p).
↪as_scaffold()
```

StrandBuilder should generally not be created directly by calling its constructor, but rather by calling the method *Design.draw_strand()*.

Although it is convenient to use chained method calls, it is also sometimes useful to assign the *StrandBuilder* object into a variable and then call the methods on that variable, particularly when creating a strand with many domains that are easiest to express in a Python loop (e.g., a long scaffold strand for a DNA origami). For example, the following code is equivalent to the above line:

```
strand_builder = design.draw_strand(0, 0)
strand_builder.to(10)
strand_builder.cross(1)
strand_builder.to(5)
strand_builder.with_modification_5p(mod.biotin_5p)
strand_builder.as_scaffold()
```

Parameters

- **design** (Design) –
- **helix** (*int*) –
- **offset** (*int*) –

cross(*helix*, *offset=None*, *move=None*)

Add crossover. To have any effect, must be followed by call to *StrandBuilder.to()* or *StrandBuilder.move()*.

Parameters

- **helix** (*int*) – *Helix* to crossover to
- **offset** (*Optional[int]*) – new offset on *helix*. If not specified, defaults to current offset. (i.e., a "vertical" crossover) Mutually excusive with *move*.
- **move** (*Optional[int]*) – Relative distance to new offset on *helix* from current offset. If not specified, defaults to using parameter *offset*. Mutually excusive with *offset*.

Returns
self

Return type
StrandBuilder

loopout(*helix*, *length*, *offset=None*, *move=None*)

Like *StrandBuilder.cross()*, but creates a *Loopout* instead of a crossover.

Parameters

- **helix** (*int*) – *Helix* to crossover to
- **length** (*int*) – length of *Loopout* to add
- **offset** (*Optional[int]*) – new offset on *helix*. If not specified, defaults to current offset. (i.e., a "vertical" loopout) Mutually excusive with *move*.
- **move** (*Optional[int]*) – Relative distance to new offset on *helix* from current offset. If not specified, defaults to using parameter *offset*. Mutually excusive with *offset*.

Returns
self

Return type
StrandBuilder

extension_3p(*num_bases*, *display_length=1.0*, *display_angle=35.0*)

Creates an *Extension* after verifying that it is valid to add an *Extension* to the *Strand* as a 3' *Extension*.

Parameters

- **num_bases** (*int*) – number of bases of *Extension* to add
- **display_length** (*float*) – display length of *Extension* to add
- **display_angle** (*float*) – display angle of *Extension* to add

Returns
self

Return type
StrandBuilder

extension_5p(*num_bases*, *display_length=1.0*, *display_angle=35.0*)

Creates an *Extension* after verifying that it is valid to add an *Extension* to the *Strand* as a 5' *Extension*.

Parameters

- **num_bases** (*int*) – number of bases of *Extension* to add
- **display_length** (*float*) – display length of *Extension* to add
- **display_angle** (*float*) – display angle of *Extension* to add

Returns
self

Return type
StrandBuilder

move(*delta*)

Extends this *StrandBuilder* on the current helix to offset given by the current offset plus *delta*, which adds a new *Domain* to the *Strand* being built. This is a "relative move", whereas *StrandBuilder.to()* and *StrandBuilder.update_to()* are "absolute moves".

NOTE: The parameter *delta* does not indicate how much we move from the current offset. It indicates the total length of the domain after the move. For instance, if we are currently on offset 10, and we call `move(5)`, this will create a domain starting at offset 10 and ending at offset 14, for a total length of 5, occuping 5 offsets: 10, 11, 12, 13, 14. (But if we imagine moving from offset 10, we've only moved by 4 offsets to arrive at 14, not 5 offsets.)

This updates the underlying *Design* with a new *Domain*, and if *StrandBuilder.loopout()* was last called on this *StrandBuilder*, also a new *Loopout*.

If two instances of *StrandBuilder.move()* are chained together, this creates two domains on the same helix. The two offsets must move in the same direction. In other words, if we call `.move(o1).move(o2)`, then `o1` and `o2` must be either both negative or both positive.

Parameters
delta (*int*) – Distance to new offset to extend to, compared to current offset. If less than current offset, the new *Domain* is reverse, otherwise it is forward.

Returns
self

Return type
StrandBuilder

to(*offset*)

Extends this *StrandBuilder* on the current helix to offset *offset*, which adds a new *Domain* to the *Strand* being built. This is an "absolute move", whereas *StrandBuilder.move()* is a "relative move".

This updates the underlying *Design* with a new *Domain*, and if *StrandBuilder.loopout()* was last called on this *StrandBuilder*, also a new *Loopout*.

If two instances of *StrandBuilder.to()* are chained together, this creates two domains on the same helix. The two offsets must move in the same direction. In other words, if the starting offset is `s`, and we call `.to(o1).to(o2)`, then either `s < o1 < o2` or `o2 < o1 < s` must be true.

To simply change the current offset after calling *StrandBuilder.to()*, without creating a new Domain, call *StrandBuilder.update_to()* instead.

Parameters
offset (*int*) – new offset to extend to. If less than current offset, the new *Domain* is reverse, otherwise it is forward.

Returns
self

Return type
StrandBuilder

update_to(*offset*)

Like *StrandBuilder.to()*, but changes the current offset without creating a new *Domain*. So unlike *StrandBuilder.to()*, several consecutive calls to *StrandBuilder.update_to()* are equivalent to only making the final call.

Generally there's no point in calling *StrandBuilder.update_to()* in one line of code. It is intended to help when a large, complex strand is being constructed in a loop.

If *StrandBuilder.cross()* or *StrandBuilder.loopout()* was just called, then *StrandBuilder.to()* and *StrandBuilder.update_to()* have the same effect.

Parameters
offset (*int*) – new offset to extend to. If less than offset of the last call to *StrandBuilder.cross()* or *StrandBuilder.loopout()*, the new *Domain* is reverse, otherwise it is forward.

Returns
self

Return type
StrandBuilder

as_circular()

Makes *Strand* being built circular.

Returns
self

Return type
StrandBuilder

as_scaffold()

Makes *Strand* being built a scaffold.

Returns
self

Return type
StrandBuilder

with_vendor_fields(*scale='25nm'*, *purification='STD'*, *plate=None*, *well=None*)

Gives *VendorFields* value to *Strand* being built.

Parameters

- **scale** (*str*) – see *VendorFields.scale*
- **purification** (*str*) – see *VendorFields.purification*
- **plate** (*Optional[str]*) – see *VendorFields.plate*
- **well** (*Optional[str]*) – see *VendorFields.well*

Returns
self

Return type
StrandBuilder

`with_modification_5p`(*mod*)

Sets Strand being built to have given 5' modification.

Parameters
mod (`Modification5Prime`) – 5' modification

Returns
self

Return type
StrandBuilder

`with_modification_3p`(*mod*)

Sets Strand being built to have given 3' modification.

Parameters
mod (`Modification3Prime`) – 3' modification

Returns
self

Return type
StrandBuilder

`with_modification_internal`(*idx*, *mod*, *warn_no_dna=True*)

Sets Strand being built to have given internal modification.

Parameters

- **idx** (*int*) – idx along DNA sequence of internal modification
- **mod** (`ModificationInternal`) – internal modification
- **warn_no_dna** (*bool*) – whether to print warning to screen if DNA has not been assigned

Returns
self

Return type
StrandBuilder

`with_color`(*color*)

Sets Strand being built to have given color.

Parameters
color (`Color`) – color to set for Strand

Returns
self

Return type
StrandBuilder

`with_sequence`(*sequence*, *assign_complement=False*)

Assigns *sequence* as DNA sequence of the *Strand* being built. This should be done after the *Strand*'s structure is done being built, e.g.,

```
design.draw_strand(0, 0).to(10).cross(1).to(5).with_sequence('AAAAAAAAAACGCGC')
```

Parameters

- **sequence** (*str*) – the DNA sequence to assign to the *Strand*
- **assign_complement** (*bool*) – whether to automatically assign the complement to existing *Strand*'s bound to this *Strand*. This has the same meaning as the parameter *assign_complement* in *Design.assign_dna()*.

Returns
self

Return type
StrandBuilder

with_domain_sequence(*sequence*, *assign_complement=False*)

Assigns *sequence* as DNA sequence of the most recently created *Domain* in the *Strand* being built. This should be called immediately after a *Domain* is created via a call to *StrandBuilder.to()*, *StrandBuilder.update_to()*, *StrandBuilder.move()*, *StrandBuilder.extension_5p()*, *StrandBuilder.extension_3p()*, or *StrandBuilder.loopout()*, e.g.,

```
design.draw_strand(0, 5)\
    .extension_5p(2).with_domain_sequence('TT')\
    .to(8).with_domain_sequence('AAA')\
    .cross(1).move(-3).with_domain_sequence('TTT')\
    .loopout(2, 4).with_domain_sequence('CCCC')\
    .to(10).with_domain_sequence('GGGGG')\
    .extension_3p(4).with_domain_sequence('AAAA')
```

Parameters

- **sequence** (*str*) – the DNA sequence to assign to the *Domain*
- **assign_complement** (*bool*) – whether to automatically assign the complement to existing *Strand*'s bound to this *Strand*. This has the same meaning as the parameter *assign_complement* in *Design.assign_dna()*.

Returns
self

Return type
StrandBuilder

with_domain_color(*color*)

Sets most recent *Domain*/*Loopout*/*Extension* to have given color.

Parameters
color (Color) – color to set for *Domain*/*Loopout*/*Extension*

Returns
self

Return type
StrandBuilder

with_name(*name*)

Assigns *name* as name of the *Strand* being built.

```
design.draw_strand(0, 0).to(10).cross(1).to(5).with_name('scaffold')
```

Parameters
name (*str*) – name to assign to the *Strand*

Returns
self

Return type
StrandBuilder

with_label(*label*)

Assigns *label* as label of the *Strand* being built.

```
design.draw_strand(0, 0).to(10).cross(1).to(5).with_label('scaffold')
```

Parameters
label (*str*) – label to assign to the *Strand*

Returns
self

Return type
StrandBuilder

with_domain_name(*name*)

Assigns *name* as of the most recently created *Domain* or *Loopout* in the *Strand* being built. This should be called immediately after a *Domain* is created via a call to *StrandBuilder.to()*, *StrandBuilder.move()*, *StrandBuilder.update_to()*, or *StrandBuilder.loopout()*, e.g.,

```
design.draw_strand(0, 0).to(10).with_domain_name('dom1*').cross(1).to(5).with_
↪domain_name('dom1')
```

Parameters
name (*str*) – name to assign to the most recently created *Domain* or *Loopout*

Returns
self

Return type
StrandBuilder

with_domain_label(*label*)

Assigns *label* as label of the most recently created *Domain* or *Loopout* in the *Strand* being built. This should be called immediately after a *Domain* or *Loopout* is created via a call to *StrandBuilder.to()*, *StrandBuilder.move()*, *StrandBuilder.update_to()*, or *StrandBuilder.loopout()*, e.g.,

```
design.draw_strand(0, 5)\
    .to(8).with_domain_label('domain 1')\
    .cross(1)\
    .to(5).with_domain_label('domain 2')\
```

(continues on next page)

(continued from previous page)

```
    .loopout(2, 4).with_domain_label('domain 3')\
    .to(10).with_domain_label('domain 4')
```

Parameters
label (*str*) – label to assign to the *Domain* or *Loopout*

Returns
self

Return type
StrandBuilder

with_deletions(*deletions*)

Assigns *deletions* as the deletion(s) of the most recently created *Domain* the *Strand* being built. This should be called immediately after a *Domain* is created via a call to *StrandBuilder.to()*, *StrandBuilder.move()*, *StrandBuilder.update_to()*, e.g.,

```
design.draw_strand(0, 0)\
    .move(8).with_deletions(4)\
    .cross(1)\
    .move(-8).with_deletions([2, 3])
```

Parameters
deletions (*Union[int, Iterable[int]]*) – a single int, or an Iterable of ints, indicating the offset at which to put the deletion(s)

Returns
self

Return type
StrandBuilder

with_insertions(*insertions*)

Assigns *insertions* as the insertion(s) of the most recently created *Domain* the *Strand* being built. This should be called immediately after a *Domain* is created via a call to *StrandBuilder.to()*, *StrandBuilder.move()*, *StrandBuilder.update_to()*, e.g.,

```
design.draw_strand(0, 0)\
    .move(8).with_insertions((4, 2))\
    .cross(1)\
    .move(-8).with_insertions([(2, 3), (3, 3)])
```

Parameters
insertions (*Union[Tuple[int, int], Iterable[Tuple[int, int]]]*) – a single pair of ints (tuple), or an Iterable of pairs of ints (tuples) indicating the offset at which to put the insertion(s)

Returns
self

Return type
StrandBuilder

class scadnano.**Strand**(*domains*, *circular=False*, *color=None*, *vendor_fields=None*, *is_scaffold=False*, *modification_5p=None*, *modification_3p=None*, *modifications_int=None*, *name=None*, *label=None*, *_helix_idx_domain_map=None*, *dna_sequence=None*)

Represents a single strand of DNA.

Each maximal portion that is continguous on a single *Helix* is a *Domain*. Crossovers from one *Helix* to another are implicitly from the 3' end of one of this Strand's *Domain*'s to the 5' end of the next *Domain*.

A portion of the *Strand* not associated to any *Helix* is represented by a *Loopout*. Two *Loopout*'s cannot occur consecutively on a *Strand*, nor can a *Strand* contain only a *Loopout* but no *Domain*.

One can set the strand to be a scaffold in the constructor:

```
import scadnano as sc

scaffold_domains = [ ... ]
scaffold_strand = sc.Strand(domains=scaffold_domains, is_scaffold=True)
```

or by calling *Strand.set_scaffold()* on the *Strand* object:

```
import scadnano as sc

scaffold_domains = [ ... ]
scaffold_strand = sc.Strand(domains=scaffold_domains)
scaffold_strand.set_scaffold()
```

or by calling *StrandBuilder.as_scaffold()* on the *StrandBuilder* object returned by *Design.strand()*:

```
import scadnano as sc

design = sc.Design(helices=[sc.Helix(max_offset=100) for _ in range(2)])
scaffold_strand = design.strand(0, 0).move(100).cross(1).move(-100).as_scaffold()
```

By default, these will give the strand the same color that cadnano uses for the scaffold.

Parameters

- **domains** (*List[Union[Domain, Loopout, Extension]]*) –
- **circular** (*bool*) –
- **color** (*Optional[Color]*) –
- **vendor_fields** (*Optional[VendorFields]*) –
- **is_scaffold** (*bool*) –
- **modification_5p** (*Optional[Modification5Prime]*) –
- **modification_3p** (*Optional[Modification3Prime]*) –
- **modifications_int** (*Dict[int, ModificationInternal]*) –
- **name** (*Optional[str]*) –
- **label** (*Optional[str]*) –
- **_helix_idx_domain_map** (*Dict[int, List[Domain]]*) –
- **dna_sequence** (*Optional[str]*) –

property dna_sequence: Optional[str]

Do not assign directly to this field. Always use *Design.assign_dna* (for complementarity checking) or *Strand.set_dna_sequence* (without complementarity checking, to allow mismatches).

Note that this does not include any vendor codes for *Modification*'s. To include those call *Strand.vendor_dna_sequence()*.

domains: List[Union[*Domain*, *Loopout*, *Extension*]]

Domain's (or *Loopout*'s or *Extension*'s) composing this *Strand*. Each *Domain* is contiguous on a single *Helix* and could be either single-stranded or double-stranded, whereas each *Loopout* and *Extension* is single-stranded and has no associated *Helix*.

circular: bool = False

If True, this *Strand* is circular and has no 5' or 3' end. Although there is still a first and last *Domain*, we interpret there to be a crossover from the 3' end of the last domain to the 5' end of the first domain, and any circular permutation of *Strand.domains* should result in a functionally equivalent *Strand*. It is illegal to have a *Modification5Prime* or *Modification3Prime* on a circular *Strand*.

color: Optional[*Color*] = None

Color to show this strand in the main view. If not specified in the constructor, a color is assigned by cycling through a list of defaults given by *ColorCycler.colors()*

vendor_fields: Optional[*VendorFields*] = None

Fields used when ordering strands from the a DNA synthesis company such as IDT (Integrated DNA Technologies, Coralville, IA). If present (i.e., not equal to `None`) then the method *Design.write_idt_bulk_input_file()* can be called to automatically generate an text file for ordering strands in test tubes: https://www.idtdna.com/site/order/oligoentry, as can the method *Design.write_idt_plate_excel_file()* for writing a Microsoft Excel file that can be uploaded to IDT's website for describing DNA sequences to be ordered in 96-well or 384-well plates: https://www.idtdna.com/site/order/plate/index/dna/1800

Currently no other vendors are supported via export methods in the package, but one could write custom export code based on these fields since most DNA synthesis companies support the same concepts of scale, purification, and a code for modifications (such as `"/5Biosg/"` for 5' biotin from IDT).

is_scaffold: bool = False

Indicates whether this *Strand* is a scaffold for a DNA origami. If any *Strand* in a *Design* is a scaffold, then the design is considered a DNA origami design.

modification_5p: Optional[*Modification5Prime*] = None

5' modification; None if there is no 5' modification. Illegal to have if *Strand.circular* is True.

modification_3p: Optional[*Modification3Prime*] = None

3' modification; None if there is no 3' modification. Illegal to have if *Strand.circular* is True.

modifications_int: Dict[int, *ModificationInternal*]

Modification's to the DNA sequence (e.g., biotin, Cy3/Cy5 fluorphores).

Maps index within DNA sequence to modification. If the internal modification is attached to a base (e.g., internal biotin, /iBiodT/ from IDT), then the index is that of the base. If it goes between two bases (e.g., internal Cy3, /iCy3/ from IDT), then the index is that of the previous base, e.g., to put a Cy3 between bases at indices 3 and 4, the index should be 3. So for an internal modified base on a sequence of length n, the allowed indices are 0,...,n-1, and for an internal modification that goes between bases, the allowed indices are 0,...,n-2.

name: Optional[str] = None

Optional name to give the strand. If specified it is shown on mouseover in the scadnano web interface.

This is used to interoperate with the dsd DNA sequence design package.

label: Optional[str] = None

This can be used to attach a "label" to associate to this *Strand*.

Useful for associating extra information with the *Strand* that will be serialized, for example, for DNA sequence design. It can be useful to create "groups" of strands related in some way.

Prior to version 0.18.0, this was allowed to be an arbitrary JSON-serializable object. Now it is just a string (see https://github.com/UC-Davis-molecular-computing/scadnano-python-package/issues/261). To store more structured data, it is necessary to serialize (convert to a string) the data manually. For example, if you want to store a list of numbers, you can do so as a string like this:

```
import json

nums = [1, 2, 3]
strand.label = json.dumps(nums)  # stores strand.label as the string '[1, 2, 3]'

# and to get the structured data back out:
nums = json.loads(strand.label)  # nums is now the list [1, 2, 3]
```

rotate_domains(*rotation*, *forward=True*)

"Rotates" the strand by replacing domains with a circular rotation, e.g., if the domains are

A, B, C, D, E, F

then `strand.rotate_domains(2)` makes the *Strand* have the same domains, but in this order:

E, F, A, B, C, D

and `strand.rotate_domains(2, forward=False)` makes

C, D, E, F, A, B

Parameters

- **rotation** (*int*) – Amount to rotate domains.
- **forward** (*bool*) – Whether to move domains forward (wrapping off 3' end back to 5' end) or backward (wrapping off 5' end back to 3' end).

Return type
None

set_scaffold(*is_scaf=True*)

Sets this *Strand* as a scaffold. Alters color to default scaffold color.

If *is_scaf* == `False`, sets this strand as not a scaffold, and leaves the color alone.

Parameters
is_scaf (*bool*) –

Return type
None

set_name(*name*)

Sets name of this *Strand*.

Parameters
name (*str*) –

Return type
None

set_label(*label*)

Sets label of this *Strand*.

Parameters
label (*Any*) –

Return type
None

set_color(*color*)

Sets color of this *Strand*.

Parameters
color (*Color*) –

Return type
None

set_circular(*circular=True*)

Sets this to be a circular *Strand* (or non-circular if optional parameter is False).

Parameters
circular (*bool*) – whether to make this *Strand* circular (True) or linear (False)

Raises
StrandError – if this *Strand* has a 5' or 3' modification

Return type
None

set_linear()

Makes this a linear (non-circular) *Strand*. Equivalent to calling *self.set_circular(False)*.

Return type
None

set_domains(*domains*)

Sets the *Domain*'s/*Loopout*'s/*Extension*'s of this *Strand* to be *domains*, which can contain a mix of *Domain*'s, *Loopout*'s, and *Extension*'s, just like the field *Strand.domains*.

Parameters
domains (*Iterable[Union[Domain, Loopout, Extension]]*) – The new sequence of *Domain*'s/*Loopout*'s/*Extension*'s to use for this *Strand*.

Raises
StrandError – if domains has two consecutive *Loopout*'s, consists of just a single *Loopout*'s or a single *Extension*, or starts or ends with a *Loopout*, or has an *Extension* on a circular *Strand*, or has an *Extension* not as the first or last element of *domains*.

Return type
None

vendor_export_name(*unique_names=False*)

Parameters
unique_names (*bool*) – If True and default name is used, enforces that strand names must be unique by encoding the forward/reverse Boolean into the name. If False (the default), uses cadnano's exact naming convention, which allows two strands to have the same default name, if they begin and end at the same (helix,offset) pair (but point in opposite directions at each). Has no effect if *Strand.vendor_fields* or *Strand.name* are defined; if those are used, they must be explicitly set to be unique.

Returns
If *Strand.name* is not None, return *Strand.name*, otherwise return the result of *Strand.default_export_name()* with parameter *unique_names*.

Return type
str

default_export_name(*unique_names=False*)

Returns a default name to use when exporting the DNA sequence. Uses cadnano's naming convention of, for example *'ST2[5]4[10]'* to indicate a strand that starts at helix 2, offset 5, and ends at helix 4, offset 10. Note that this naming convention is not unique: two strands in the system could share this name. To ensure it is unique, set the parameter *unique_names* to True, which will modify the name with forward/reverse information from the first domain that uniquely identifies the strand, e.g., *'ST2[5]F4[10]'* or *'ST2[5]R4[10]'*.

If the strand is a scaffold (i.e., if *Strand.is_scaffold* is True), then the name will begin with *'SCAF'* instead of *'ST'*.

Parameters
unique_names (*bool*) – If True, enforces that strand names must be unique by encoding the forward/reverse Boolean into the name. If False (the default), uses cadnano's exact naming convention, which allows two strands to have the same default name, if they begin and end at the same (helix,offset) pair (but point in opposite directions at each).

Returns
default name to export (used, for example, by idt DNA export methods *Design.write_idt_plate_excel_file()* and *Design.write_idt_bulk_input_file()* if *Strand.name* and `Strand.vendor_fields.name` are both not set)

Return type
str

set_modification_5p(*mod*)

Sets 5' modification to be *mod*. *Strand.circular* must be False.

Parameters
mod (Modification5Prime) –

Return type
None

set_modification_3p(*mod*)

Sets 3' modification to be *mod*. *Strand.circular* must be False.

Parameters
mod (Modification3Prime) –

Return type
None

remove_modification_5p()

Removes 5' modification.

Return type
None

remove_modification_3p()

Removes 3' modification.

Return type
None

set_modification_internal(*idx*, *mod*, *warn_on_no_dna=True*)

Adds internal modification *mod* at given DNA index *idx*.

Parameters

- **idx** (*int*) –
- **mod** (ModificationInternal) –
- **warn_on_no_dna** (*bool*) –

Return type
None

remove_modification_internal(*idx*)

Removes internal modification at given DNA index *idx*.

Parameters
idx (*int*) –

Return type
None

first_domain()

First domain on this *Strand*.

Return type
Domain

last_domain()

Last domain on this *Strand*.

Return type
Domain

dna_sequence_delimited(*delimiter*)

Parameters
delimiter (*str*) – string to put in between DNA sequences of each domain

Returns
DNA sequence of this *Strand*, with *delimiter* in between DNA sequences of each *Domain* or *Loopout*.

Return type
str

set_dna_sequence(*sequence*)

Set this *Strand*'s DNA sequence to *seq* WITHOUT checking for complementarity with overlapping *Strand*'s or automatically assigning their sequences. To assign a sequence to a *Strand* and have the overlapping *Strand*'s automatically have the appropriate Watson-Crick complements assigned, use *Design.assign_dna*.

All whitespace in *sequence* is removed, and lowercase bases 'a', 'c', 'g', 't' are converted to uppercase.

sequence, after all whitespace is removed, must be exactly the same length as *Strand.dna_length()*. Wildcard symbols (DNA_case_wildcard) are allowed to leave part of the DNA unassigned.

Parameters
sequence (*str*) –

Return type
None

dna_length()

Return sum of DNA length of *Domain*'s and *Loopout*'s of this *Strand*.

Return type
int

bound_domains()

Domain's of this *Strand* that are not *Loopout*'s.

Return type
List[Domain]

offset_5p()

5' offset of this entire *Strand*, INCLUSIVE.

Return type
int

offset_3p()

3' offset of this entire *Strand*, INCLUSIVE.

Return type
int

overlaps(*other*)

Indicates whether *self* overlaps *other_strand*, meaning that the set of offsets occupied by *self* has nonempty intersection with those occupied by *other_strand*.

Parameters
other (Strand) –

Return type
bool

assign_dna_complement_from(*other*)

Assuming a DNA sequence has been assigned to *other*, assign its Watson-Crick complement to the portions of this Strand that are bound to *other*.

Generally this is not called directly; use *Design.assign_dna()* to assign a DNA sequence to a *Strand*. The method *Design.assign_dna()* will calculate which other *Strand*'s need to be assigned via *Strand.assign_dna_complement_from()*.

However, it is permitted to assign the field *Strand.dna_sequence* directly via the method *Strand.set_dna_sequence()*. This is used, for instance, to assign a DNA sequence to a *Strand* bound to another *Strand* with an assigned DNA sequence where they overlap. In this case no error checking about sequence complementarity is done. This can be used to intentionally assign *mismatching* DNA sequences to *Strand*'s that are bound on a *Helix*.

Parameters
other (Strand) –

Return type
None

dna_index_start_domain(*domain*)

Returns index in DNA sequence of domain, e.g., if there are five domains

012 3 45 678 9 AAA-C-GG-TTT-ACGT

Then their indices, respectively in order, are 0, 3, 4, 6, 9.

Parameters
domain (*Union[Domain, Loopout]*) –

any
to find the start DNA index of

Returns
index (within DNA sequence string) of substring of DNA starting with given *Domain*

Return type
int

first_bound_domain()

First *Domain* (i.e., not a *Loopout*) on this *Strand*.

Currently the first and last strand must not be *Loopout*'s, so this should return the same domain as *Strand.first_domain()*, but in case an initial or final *Loopout* is supported in the future, this method is provided.

Return type
Domain

last_bound_domain()

Last *Domain* (i.e., not a *Loopout*) on this *Strand*.

Currently the first and last strand must not be *Loopout*'s, so this should return the same domain as *Strand.first_domain()*, but in case an initial or final *Loopout* is supported in the future, this method is provided.

Return type
Domain

reverse()

Reverses "polarity" of this *Strand*.

Does NOT check whether this keeps the *Design* legal, so be cautious in calling this method directly. To reverse every *Strand*, called *Design.reverse_all()*. If the design was legal before, it will be legal after calling that method.

Return type
None

vendor_dna_sequence(*domain_delimiter=''*)

Parameters
domain_delimiter (*str*) – string to put in between DNA sequences of each domain, and between 5'/3' modifications and DNA. Note that the delimiter is not put between internal modifications and the next base(s) in the same domain.

Returns
DNA sequence as it needs to be typed to order from a DNA synthesis vendor, with *Modification5Prime*'s, *Modification3Prime*'s, and *ModificationInternal*'s represented with text codes, e.g., for IDT DNA, using "/5Biosg/ACGT" for sequence ACGT with a 5' biotin modification.

Return type
str

no_modifications_version()

Returns
version of this *Strand* with no DNA modifications.

Return type
Strand

class scadnano.**StrandOrder**(*value*)

Which part of a *Strand* to use for sorting in the key function returned by *strand_order_key_function()*.

five_prime = 0

5' end of the strand

three_prime = 1

3' end of the strand

five_or_three_prime = 2

Either 5' end or 3' end is used, whichever is first according to the sort order.

top_left_domain = 3

The start offset of the "top-left" *Domain* of the *Strand*: the *Domain* whose *Domain.helix* is minimal, and, among all such *Domain*'s, the one with minimal *Domain.start*.

scadnano.**strand_order_key_function**(*, *column_major=True*, *strand_order*)

Returns a key function indicating a sorted order for *Strand*'s. Useful as a parameter for `Design.()`.

Parameters

- **column_major** (*bool*) – If true, column major order is used: ordered by base offset first, then by helix. Otherwise row-major order is used: ordered by helix first, then by base offset.
- **strand_order** (StrandOrder) – Which part of the strand to use as a key for the sorted order. See *StrandOrder* for definitions.

Returns

A key function that can be passed to `Design.()` to specify a sorted order for the *Strand*'s.

Return type
Callable[[Strand], *Any*]

exception scadnano.**IllegalDesignError**(*the_cause*)

Indicates that some aspect of the *Design* object is illegal.

Parameters
the_cause (*str*) –

Return type
None

exception scadnano.**StrandError**(*strand*, *the_cause*)

Indicates that the *Design* is illegal due to some specific *Strand*. Information about the *Strand* is embedded in the error message when this exception is raised that helps to identify which *Strand* caused the problem.

Parameters

- **strand** (Strand) –
- **the_cause** (*str*) –

Return type
None

class scadnano.**PlateType**(*value*)

Represents two different types of plates in which DNA sequences can be ordered.

wells96 = 96

96-well plate.

wells384 = 384

384-well plate.

num_wells_per_plate()

Returns
number of wells in this plate type

Return type
int

min_wells_per_plate()

Returns
minimum number of wells in this plate type to avoid extra charge by IDT

Return type
int

class scadnano.**PlateMap**(*plate_name*, *plate_type*, *well_to_strand*)

Represents a "plate map", i.e., a drawing of a 96-well or 384-well plate, indicating which subset of wells in the plate have strands. It is an intermediate representation of structured data about the plate map that is converted to a visual form, such as Markdown, via the export_* methods.

Parameters

- **plate_name** (*str*) –
- **plate_type** (PlateType) –
- **well_to_strand** (*Dict[str, Strand]*) –

plate_name: str

Name of this plate.

plate_type: *PlateType*

Type of this plate (96-well or 384-well).

well_to_strand: Dict[str, *Strand*]

dictionary mapping the name of each well (e.g., "C4") to the strand in that well.

Wells with no strand in the PlateMap are not keys in the dictionary.

to_table(*well_marker=None*, *title_level=3*, *warn_unsupported_title_format=True*, *vertical_borders=False*, *tablefmt='pipe'*, *stralign='default'*, *missingval=''*, *showindex='default'*, *disable_numparse=False*, *colalign=None*)

Exports this plate map to string format, with a header indicating information such as the plate's name and volume to pipette. By default the text format is Markdown, which can be rendered in a jupyter notebook using `display` and `Markdown` from the package IPython.display:

```
plate_maps = design.plate_maps()
maps_strs = '\n\n'.join(plate_map.to_table() for plate_map in plate_maps)
from IPython.display import display, Markdown
display(Markdown(maps_strs))
```

Markdown format is used by default, generating a string such as this:

```
plate "5 monomer synthesis"

|     | 1    | 2      | 3      | 4    | 5         | 6   | 7   | 8   | 9   | 10  ␣
↪| 11   | 12   |
|-----|------|--------|--------|------|-----------|-----|-----|-----|-----|------
↪|------|------|
| A   | mon0 | mon0_F |        | adp0 |           |     |     |     |     |     ␣
↪|      |      |
| B   | mon1 | mon1_Q | mon1_F | adp1 | adp_sst1 |     |     |     |     |     ␣
↪|      |      |
| C   | mon2 | mon2_F | mon2_Q | adp2 | adp_sst2 |     |     |     |     |     ␣
↪|      |      |
| D   | mon3 | mon3_Q | mon3_F | adp3 | adp_sst3 |     |     |     |     |     ␣
↪|      |      |
| E   | mon4 |        | mon4_Q | adp4 | adp_sst4 |     |     |     |     |     ␣
↪|      |      |
| F   |      |        |        | adp5 |           |     |     |     |     |     ␣
↪|      |      |
| G   |      |        |        |      |           |     |     |     |     |     ␣
↪|      |      |
| H   |      |        |        |      |           |     |     |     |     |     ␣
↪|      |      |
```

or, with the *PlateMap.to_table()* parameter *well_marker* set to `'*'` (in case you don't need to see the strand names and just want to see which wells are marked):

```
|     | 1   | 2   | 3   | 4   | 5   | 6   | 7   | 8   | 9   | 10   | 11   | 12 ␣
↪ |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|------|------|-----
↪-|
| A   | *   | *   |     | *   |     |     |     |     |     |      |      |    ␣
↪ |
| B   | *   | *   | *   | *   | *   |     |     |     |     |      |      |    ␣
↪ |
| C   | *   | *   | *   | *   | *   |     |     |     |     |      |      |    ␣
↪ |
| D   | *   | *   | *   | *   | *   |     |     |     |     |      |      |    ␣
↪ |
| E   | *   |     | *   | *   | *   |     |     |     |     |      |      |    ␣
↪ |
| F   |     |     |     | *   |     |     |     |     |     |      |      |    ␣
↪ |
| G   |     |     |     |     |     |     |     |     |     |      |      |    ␣
↪ |
| H   |     |     |     |     |     |     |     |     |     |      |      |    ␣
↪ |
```

If *well_marker* is not specified, then each strand must have a name. *well_marker* can also be a function of the well; for instance, if it is the identity function `lambda x:x`, then each well has its own address as the entry:

```
|     | 1   | 2   | 3   | 4   | 5   | 6   | 7   | 8   | 9   | 10   | 11   | 12 ␣
↪ |
```

(continues on next page)

(continued from previous page)

```
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|------|------|-----
↪-|
| A   | A1  | A2  |     | A4  |     |     |     |     |     |      |      |  ␣
↪  |
| B   | B1  | B2  | B3  | B4  | B5  |     |     |     |     |      |      |  ␣
↪  |
| C   | C1  | C2  | C3  | C4  | C5  |     |     |     |     |      |      |  ␣
↪  |
| D   | D1  | D2  | D3  | D4  | D5  |     |     |     |     |      |      |  ␣
↪  |
| E   | E1  |     | E3  | E4  | E5  |     |     |     |     |      |      |  ␣
↪  |
| F   |     |     |     | F4  |     |     |     |     |     |      |      |  ␣
↪  |
| G   |     |     |     |     |     |     |     |     |     |      |      |  ␣
↪  |
| H   |     |     |     |     |     |     |     |     |     |      |      |  ␣
↪  |
```

This method uses the Python tabulate package (https://pypi.org/project/tabulate/). The parameters are identical to that of the *tabulate* function and are passed along to it, except for *tabular_data* and *headers*, which are computed from this plate map. In particular, the parameter *tablefmt* has default value *'pipe'*, which creates a Markdown format. To create other formats such as HTML, change the value of *tablefmt*; see https://github.com/astanin/python-tabulate#readme for other possible formats.

Parameters

- **well_marker** (*Optional[Union[str, Callable[[str], str]]]*) – By default the strand's name is put in the relevant plate entry. If *well_marker* is specified and is a string, then that string is put into every well with a strand in the plate map instead. This is useful for printing plate maps that just put, for instance, an *'X'* in the well to pipette (e.g., specify `well_marker='X'`), e.g., for experimental mixes that use only some strands in the plate. To enable the string to depend on the well position (instead of being the same string in every well), *well_marker* can also be a function that takes as input a string representing the well (such as `"B3"` or `"E11"`), and outputs a string. For example, giving the identity function `mix.to_table(well_marker=lambda x: x)` puts the well address itself in the well.
- **title_level** (*int*) – The "title" is the first line of the returned string, which contains the plate's name and volume to pipette. The *title_level* controls the size, with 1 being the largest size, (header level 1, e.g., # title in Markdown or <h1>title</h1> in HTML) and 6 being the smallest size.
- **warn_unsupported_title_format** (*bool*) – If True, prints a warning if *tablefmt* is a currently unsupported option for the title. The currently supported formats for the title are 'github', 'html', 'unsafehtml', 'rst', 'latex', 'latex_raw', 'latex_booktabs', "latex_longtable". If *tablefmt* is another valid option, then the title will be the Markdown format, i.e., same as for *tablefmt* = 'github'.
- **vertical_borders** (*bool*) – If true, then *tablefmt* must be set to *html* or *unsafehtml*, and the returned HTML will use inline styles to ensure there are vertical borders between columns of the table. The vertical borders make it easier to see which column a well is in. This is useful when rendering in a Jupyter notebook, since the inline styles will be preserved when saving the Jupyter notebook using the nbconvert tool: https://nbconvert.readthedocs.io/en/latest/

- **tablefmt** (*str*) – By default set to *'pipe'* to create a Markdown table. For other options see https://github.com/astanin/python-tabulate#readme
- **stralign** (*str*) – See https://github.com/astanin/python-tabulate#readme
- **missingval** (*str*) – See https://github.com/astanin/python-tabulate#readme
- **showindex** (*str*) – See https://github.com/astanin/python-tabulate#readme
- **disable_numparse** (*bool*) – See https://github.com/astanin/python-tabulate#readme
- **colalign** (*Optional[bool]*) – See https://github.com/astanin/python-tabulate#readme

Returns
a string representation of this plate map

Return type
str

scadnano.**bases_complementary**(*base1*, *base2*, *allow_wildcard=False*, *allow_none=False*)

Indicates if *base1* and *base2* are complementary DNA bases.

Parameters

- **base1** (*str*) – first DNA base
- **base2** (*str*) – second DNA base
- **allow_wildcard** (*bool*) – if true a "wildcard" (the symbol '?') is considered to be complementary to anything
- **allow_none** (*bool*) – if true the object None is considered to be complementary to anything

Returns
whether *base1* and *base2* are complementary DNA bases

Return type
bool

scadnano.**reverse_complementary**(*seq1*, *seq2*, *allow_wildcard=False*, *allow_none=False*)

Indicates if *seq1* and *seq2* are reverse complementary DNA sequences.

Parameters

- **seq1** (*str*) – first DNA sequence
- **seq2** (*str*) – second DNA sequence
- **allow_wildcard** (*bool*) – if true a "wildcard" (the symbol '?') is considered to be complementary to anything
- **allow_none** (*bool*) – if true the object None is considered to be complementary to anything

Returns
whether *seq1* and *seq2* are reverse complementary DNA sequences

Return type
bool

class scadnano.**Design**(*, *helices=None*, *groups=None*, *strands=None*, *grid=Grid.none*, *helices_view_order=None*, *geometry=None*)

Object representing the entire design of the DNA structure.

Parameters

- **helices** (*Dict[int, Helix]*) –

- **groups** (*Dict[str, HelixGroup]*) –
- **strands** (*List[Strand]*) –
- **grid** (*Grid*) –
- **helices_view_order** (*List[int]*) –
- **geometry** (*Geometry*) –

automatically_assign_color: bool = True

If *automatically_assign_color* = `False`, then for any *Strand* such that *Strand.color* = `None`, do not automatically assign a *Color* to it. In this case color will be set to its default of `None` and will not be written to the JSON with *Design.write_scadnano_file()* or *Design.to_json()*.

strands: List[*Strand*]

All of the *Strand*'s in this *Design*.

Required field.

geometry: *Geometry*

Controls some geometric/physical aspects of this *Design*.

groups: Dict[str, *HelixGroup*] = None

HelixGroup's in this *Design*.

helices: Dict[int, *Helix*] = None

All of the *Helix*'s in this *Design*. This is a dictionary mapping index to the *Helix* with that index; if helices have indices 0, 1, …, num_helices-1, then this can be used as a list of Helices.

Optional field. If not specified, then the number of helices will be just large enough to store the largest index *Domain.helix* stored in any *Domain* in *Design.strands*.

property helices_view_order: List[int]

Return helices_view_order of this *Design* if no *HelixGroup*'s are being used, otherwise raise a ValueError.

Returns
helices_view_order of this *Design*

property grid: *Grid*

Return grid of this *Design* if no *HelixGroup*'s are being used, otherwise raise a ValueError.

Returns
grid of this *Design*

set_grid(*grid*)

Sets the grid of the default *HelixGroup*, if the default is being used, otherwise raises an exception.

Parameters
grid (*Grid*) – new grid to set for the (only) *HelixGroup* in this *Design*

Raises
IllegalDesignError – if there is more than one *HelixGroup* in this *Design*

Return type
None

helices_idxs_in_group(*group_name*)

Indexes of *Helix*'s in this group. Must be associated with a *Design* for this to work.

Parameters
group_name (*str*) – name of group

Returns
list of indices of *Helix*'s in this *HelixGroup*

Return type
List[int]

`pitch_of_helix`(*helix*)

Same as the pitch of helix's *HelixGroup*

Parameters
helix (Helix) –

Return type
float

`yaw_of_helix`(*helix*)

Same as the yaw of helix's *HelixGroup*

Parameters
helix (Helix) –

Return type
float

`roll_of_helix`(*helix*)

Roll of helix's *HelixGroup* plus *Helix.roll*

Parameters
helix (Helix) –

Return type
float

`static from_scadnano_file`(*filename*)

Loads a *Design* from the file with the given name.

Parameters
filename (*str*) – name of the file with the design. Should be a JSON file ending in .sc

Returns
Design described in the file

Return type
Design

`static from_scadnano_json_str`(*json_str*)

Loads a *Design* from the given JSON string.

Parameters
json_str (*str*) – JSON description of the *Design*

Returns
Design described in the JSON string

Return type
Design

`static from_scadnano_json_map`(*json_map*)

Loads a *Design* from the given JSON object (i.e., Python object obtained by calling json.loads(json_str) from a string representing contents of a JSON file.

Parameters
json_map (*dict*) – map describing the *Design*; should be JSON serializable via `encode(json_map)`

Returns
Design described in the object

Return type
Design

property scaffold: Optional[*Strand*]

Returns the first scaffold in this *Design*, if there is one, or `None` otherwise.

base_pairs(*allow_mismatches=False*)

Base pairs in this design, represented as a dict mapping a *Helix.idx* to a list of offsets on that helix where two strands are.

If a *Helix* has no base pairs, then its *Helix.idx* is not a key in the returned dict.

An offset with a deletion on either *Domain* is not considered a base pair.

Insertions are more complex. If *allow_mismatches* is False, then an offset with an insertion on *both* *Domain*'s is considered a *single* base pair so long as the DNA sequences on each insertion are the same length and complementary. If *allow_mismatches* is True then an offset with an insertion on *either* *Domain*'s is considered a *single* base pair regardless of the length or DNA sequences of either insertion.

To calculate "true" base pairs in the presence of deletions and insertions, it is recommended first to remove the deletions and insertions using the method *Design.inline_deletions_insertions()*.

Parameters
allow_mismatches (*bool*) – if True, then all offsets on a *Helix* where there is both a forward and reverse *Domain* will be included. Otherwise, only offsets where the *Domain*'s have complementary bases will be included.

Returns
dict mapping each helix_idx to a list of offsets on that helix where the base pairs are

Return type
Dict[int, *List*[int]]

static from_cadnano_v2(*directory=''*, *filename=None*, *json_dict=None*)

Creates a Design from a cadnano v2 design. The design can either be specified as a filename (assumed to be the a JSON file containing the cadnano design) or as a Python dictionary, assumed to be the result of importing the JSON from the cadnano file.

Exactly one of *filename* or *json_dict* should be specified.

Parameters

- **directory** (*str*) – directory in which to look for *filename*; current directory by default. Ignored if *json_dict* is specified.
- **filename** (*Optional[str]*) – name of file containing cadnano design. Mutually exclusive with *json_dict*.
- **json_dict** (*Optional[dict]*) – cadnano design represented as a Python dict. (assumed to be the result of json.load on a cadnano file)

Returns
An scadnano design equivalent to the specified cadnano design.

Return type
Design

plate_maps(*warn_duplicate_strand_names=True*, *plate_type=PlateType.wells96*, *strands=None*)

Returns a list of *PlateMap*'s from this *Design*. Each *PlateMap* can be exported to a string, in Markdown format by default, by calling *PlateMap.to_table()*, generating a string such as this:

```
plate "5 monomer synthesis"

|     | 1    | 2      | 3      | 4    | 5         | 6   | 7   | 8   | 9   | 10  | 11   | 12   |
|-----|------|--------|--------|------|-----------|-----|-----|-----|-----|-----|------|------|
| A   | mon0 | mon0_F |        | adp0 |           |     |     |     |     |     |      |      |
| B   | mon1 | mon1_Q | mon1_F | adp1 | adp_sst1  |     |     |     |     |     |      |      |
| C   | mon2 | mon2_F | mon2_Q | adp2 | adp_sst2  |     |     |     |     |     |      |      |
| D   | mon3 | mon3_Q | mon3_F | adp3 | adp_sst3  |     |     |     |     |     |      |      |
| E   | mon4 |        | mon4_Q | adp4 | adp_sst4  |     |     |     |     |     |      |      |
| F   |      |        |        | adp5 |           |     |     |     |     |     |      |      |
| G   |      |        |        |      |           |     |     |     |     |     |      |      |
| H   |      |        |        |      |           |     |     |     |     |     |      |      |
```

See the documentation for *PlateMap.to_table()* for more information on configuring the returned string format.

All *Strand*'s in the design that have a field *Strand.vendor_fields* with `Strand.vendor_fields.plate` specified are included in some returned *PlateMap*. The number of *PlateMap*'s in the returned list is equal to the number of different plate names specified across all *Strand*'s in the design.

If parameter *strands* is given, then a subset of strands is included. This is useful for specifying a mix of strands for a particular experiment, which come from a plate but does not include every strand in the plate.

Parameters

- **warn_duplicate_strand_names** (*bool*) – If True, prints a warning to the screen if multiple *Strand*'s exist with the same value for *Strand.name*.
- **plate_type** (PlateType) – Type of plate: 96 or 384 well.
- **strands** (*Optional[Iterable[*Strand*]]*) – If specified, only the *Strand*'s in *strands* are put in the *PlateMap*.

Returns

list of *PlateMap*'s for *Strand*'s in this design with IDT plates specified; length of list is equal to number of unique plate names among all *Strand*'s in this design

Return type

List[PlateMap]

modifications(*mod_type=None*)

Returns either set of all modifications in this *Design*, or set of all modifications of a given type (5', 3', or internal).

Parameters
mod_type (*Optional[ModificationType]*) – type of modifications (5', 3', or internal); if not specified, all three types are returned

Returns
Set of all modifications in this *Design* (possibly of a given type).

Return type
Set[Modification]

draw_strand(*helix*, *offset*)

Used for chained method building the *Strand* domain by domain, in order from 5' to 3'. For example

```
design.draw_strand(0, 7).to(10).cross(1).to(5).cross(2).to(15)
```

This creates a *Strand* in this *Design* equivalent to

```
design.add_strand(Strand([
    sc.Domain(0, True, 7, 10),
    sc.Domain(1, False, 5, 10),
    sc.Domain(2, True, 5, 15),
]))
```

Loopouts can also be included:

```
design.draw_strand(0, 7).to(10).cross(1).to(5).loopout(2, 3).to(15)
```

This creates a *Strand* in this *Design* equivalent to

```
design.add_strand(Strand([
    sc.Domain(0, True, 7, 10),
    sc.Domain(1, False, 5, 10),
    sc.Loopout(3),
    sc.Domain(2, True, 5, 15),
]))
```

Each call to *Design.draw_strand()*, *StrandBuilder.cross()*, *StrandBuilder.loopout()*, *StrandBuilder.to()* *StrandBuilder.move()* *StrandBuilder.update_to()*, returns a *StrandBuilder* object.

Each call to *StrandBuilder.to()*, *StrandBuilder.move()*, *StrandBuilder.update_to()*, or *StrandBuilder.loopout()* modifies the *Design* by replacing the Strand with an updated version.

See the documentation for *StrandBuilder* for the methods available to call in this way.

Parameters

- **helix** (*int*) – starting *Helix*
- **offset** (*int*) – starting offset on *helix*

Returns
StrandBuilder object representing the partially completed *Strand*

Return type
StrandBuilder

strand(*helix*, *offset*)

Same functionality as *Design.draw_strand()*.

Deprecated since version 0.17.2: Use *Design.draw_strand()* instead, which is a better name. This method will be removed in a future version.

Parameters

- **helix** (*int*) –
- **offset** (*int*) –

Return type
StrandBuilder

assign_m13_to_scaffold(*rotation=5587*, *variant=M13Variant.p7249*)

Assigns the scaffold to be the sequence of M13: *m13()* with the given *rotation* and *M13Variant*.

Raises *IllegalDesignError* if the number of scaffolds is not exactly 1.

Parameters

- **rotation** (*int*) – rotation of M13 to use. See *m13()* for explanation.
- **variant** (M13Variant) – which variant of M13 to use. See *M13Variant*.

Return type
None

to_cadnano_v2_serializable(*name=''*)

Converts the design to a JSON-serializable Python object (a dict) representing the cadnano v2 format. Calling json.dumps on this object will result in a string representing the cadnano c2 format; this is essentially what is done in *Design.to_cadnano_v2_json()*.

Please see the spec https://github.com/UC-Davis-molecular-computing/scadnano-python-package/blob/main/misc/cadnano-format-specs/v2.txt for more info on that format.

Parameters
name (*str*) – Name of the design.

Returns
a Python dict representing the cadnano v2 format for this *Design*

Return type
Dict[str, *Any*]

to_cadnano_v2_json(*name=''*, *whitespace=True*)

Converts the design to the cadnano v2 format.

Please see the spec https://github.com/UC-Davis-molecular-computing/scadnano-python-package/blob/main/misc/cadnano-format-specs/v2.txt for more info on that format.

If the cadnano file is intended to be used with CanDo (https://cando-dna-origami.org/), the optional parameter *whitespace* must be set to False.

Parameters

- **name** (*str*) – Name of the design.
- **whitespace** (*bool*) – Whether to include whitespace in the exported file. Set to False to use this with CanDo (https://cando-dna-origami.org/), since that tool generates an error if the cadnano file contains whitespace.

Returns
a string in the cadnano v2 format representing this *Design*

Return type
str

set_helices_view_order(*helices_view_order*)

Sets helices_view_order.

Parameters
helices_view_order (*List[int]*) – new view order of helices

Return type
None

strands_starting_on_helix(*helix*)

Return list of *Strand*'s that begin (have their 5' end) on the *Helix* with index *helix*.

Parameters
helix (*int*) –

Return type
List[Strand]

strands_ending_on_helix(*helix*)

Return list of *Strand*'s that finish (have their 3' end) on the *Helix* with index *helix*.

Parameters
helix (*int*) –

Return type
List[Strand]

domain_at(*helix*, *offset*, *forward*)

Return *Domain* that overlaps *offset* on helix with idx *helix* and has *Domain.forward* = True, or None if there is no such *Domain*.

Parameters

- **helix** (*int*) – TODO
- **offset** (*int*) – TODO
- **forward** (*bool*) – TODO

Returns
TODO

Return type
Optional[Domain]

domains_at(*helix*, *offset*)

Return list of *Domain*'s that overlap *offset* on helix with idx *helix*.

If constructed properly, this list should have 0, 1, or 2 elements.

Parameters

- **helix** (*int*) –
- **offset** (*int*) –

Return type
List[Domain]

add_strand(*strand*)

Add *strand* to this design.

Parameters
strand (Strand) –

Return type
None

remove_strand(*strand*)

Remove *strand* from this design.

Parameters
strand (Strand) –

Return type
None

append_domain(*strand*, *domain*)

Same as *Design.insert_domain*, but inserts at end.

Parameters

- **strand** (Strand) – strand to append *domain* to
- **domain** (*Union[Domain, Loopout, Extension]*) – *Domain* or *Loopout* to append to *Strand*

Return type
None

insert_domain(*strand*, *order*, *domain*)

Insert *Domain* into *strand* at index given by *order*. Uses same indexing as Python lists, e.g., `design.insert_domain(strand, domain, 0)` inserts `domain` as the new first *Domain*.

Parameters

- **strand** (Strand) –
- **order** (*int*) –
- **domain** (*Union[Domain, Loopout, Extension]*) –

Return type
None

remove_domain(*strand*, *domain*)

Remove *Domain* from *strand*.

Parameters

- **strand** (Strand) –
- **domain** (*Union[Domain, Loopout]*) –

Return type
None

to_json(*suppress_indent=True*)

Return string representing this Design, suitable for reading by scadnano if written to a JSON file ending in extension *default_scadnano_file_extension*.

Parameters
suppress_indent (*bool*) – whether to suppress indenting JSON for "small" objects such as short lists, e.g., grid coordinates. If True, something like this will be written:

```
{
  "grid_position": [1, 2]
}
```

instead of this:

```
{
  "grid_position": [
    1,
    2
  ]
}
```

Return type
str

add_deletion(*helix*, *offset*)

Adds a deletion to every *scadnano.Strand* at the given helix and base offset.

Parameters

- **helix** (*int*) –
- **offset** (*int*) –

Return type
None

add_insertion(*helix*, *offset*, *length*)

Adds an insertion with the given length to every *scadnano.Strand* at the given helix and base offset, with the given length.

Parameters

- **helix** (*int*) –
- **offset** (*int*) –
- **length** (*int*) –

Return type
None

set_start(*domain*, *start*)

Sets `Domain.start` to *start*.

Parameters

- **domain** (Domain) –
- **start** (*int*) –

Return type
None

set_end(*domain*, *end*)

Sets `Domain.end` to *end*.

Parameters

- **domain** (Domain) –
- **end** (*int*) –

Return type
None

move_strand_offsets(*delta*)

Moves all strands backward (if *delta* < 0) or forward (if *delta* > 0) by *delta*.

Parameters
delta (*int*) –

Return type
None

move_strands_on_helices(*delta*)

Moves all strands up (if *delta* < 0) or down (if *delta* > 0) by the number of helices given by *delta*.

Parameters
delta (*int*) –

Return type
None

assign_dna(*strand*, *sequence*, *assign_complement=True*, *domain=None*, *check_length=False*)

Assigns *sequence* as DNA sequence of *strand*.

If any *scadnano.Strand* is bound to *strand*, it is assigned the reverse Watson-Crick complement of the relevant portion, and any remaining portions of the other strand that have not already been assigned a DNA sequence are assigned to be the symbol *DNA_base_wildcard*.

Before assigning, *sequence* is first forced to be the same length as *strand* as follows: If *sequence* is longer, it is truncated. If *sequence* is shorter, it is padded with *DNA_base_wildcard*'s. This can be disabled by setting *check_length* to True, in which case the method raises an *IllegalDesignError* if the lengths do not match.

All whitespace in *sequence* is removed, and lowercase bases 'a', 'c', 'g', 't' are converted to uppercase.

Parameters

- **strand** (Strand) – *Strand* to assign DNA sequence to
- **sequence** (*str*) – string of DNA bases to assign
- **assign_complement** (*bool*) – Whether to assign the complement DNA sequence to any *Strand* that is bound to this one (default True)
- **domain** (*Optional[Union[*Domain, Loopout, Extension*]]*) – *Domain* on *strand* to assign. If None, then the whole *Strand* is given a DNA sequence. Otherwise, only *domain* is assigned, and the rest of the *Domain*'s on *strand* are left alone (either keeping their DNA sequence, or being assigned *DNA_base_wildcard* if no DNA sequence was previously assigned.) If *domain* is specified, then `len(sequence)` must be least than or equal to the number of bases on *domain*. (i.e., `domain.dna_length()`)
- **check_length** (*bool*) – If True, raises *IllegalDesignError* if length of *Strand* or *Domain* being assigned to does not match the length of the DNA sequence.

Raises
IllegalDesignError – If *check_length* is True and the length of *Strand* or *Domain* being assigned to does not match the length of the DNA sequence.

Return type
None

to_idt_bulk_input_format(*delimiter=','*, *domain_delimiter=''*, *key=None*, *warn_duplicate_name=False*, *only_strands_with_idt=False*, *export_scaffold=False*, *export_non_modified_strand_version=False*)

Called by *Design.write_idt_bulk_input_file()* to determine what string to write to the file. This function can be used to get the string directly without creating a file.

Parameters have the same meaning as in *Design.write_idt_bulk_input_file()*.

Returns
string that is written to the file in the method *Design.write_idt_bulk_input_file()*.

Parameters

- **delimiter** (*str*) –
- **domain_delimiter** (*str*) –
- **key** (*Optional[Callable[[Strand], Any]]*) –
- **warn_duplicate_name** (*bool*) –
- **only_strands_with_idt** (*bool*) –
- **export_scaffold** (*bool*) –
- **export_non_modified_strand_version** (*bool*) –

Return type
str

write_idt_bulk_input_file(*, *directory='.'*, *filename=None*, *key=None*, *extension=None*, *delimiter=','*, *domain_delimiter=''*, *warn_duplicate_name=True*, *only_strands_with_idt=False*, *export_scaffold=False*, *export_non_modified_strand_version=False*)

Write `.idt` text file encoding the strands of this *Design* with the field *Strand.vendor_fields*, suitable for pasting into the "Bulk Input" field of IDT (Integrated DNA Technologies, Coralville, IA, https://www.idtdna.com/), with the output file having the same name as the running script but with `.py` changed to `.idt`, unless *filename* is explicitly specified. For instance, if the script is named `my_origami.py`, then the sequences will be written to `my_origami.idt`. If *filename* is not specified but *extension* is, then that extension is used instead of `idt`. At least one of *filename* or *extension* must be `None`.

The string written is that returned by *Design.to_idt_bulk_input_format()*.

Parameters

- **directory** (*str*) – specifies a directory in which to place the file, either absolute or relative to the current working directory. Default is the current working directory.
- **filename** (*Optional[str]*) – optional custom filename to use (instead of currently running script)
- **key** (*Optional[Callable[[Strand], Any]]*) – key function used to determine order in which to output strand sequences. Some useful defaults are provided by *strand_order_key_function()*
- **extension** (*Optional[str]*) – alternate filename extension to use (instead of idt)
- **delimiter** (*str*) – symbol to delimit the four IDT fields name,sequence,scale,purification.
- **domain_delimiter** (*str*) – This is placed between the DNA sequences of adjacent domains on a strand. For instance, IDT (Integrated DNA Technologies, Coralville, IA, https://www.idtdna.com/) ignores spaces, so setting *domain_delimiter* to `' '` will insert a space between adjacent domains while remaining readable by IDT's website.

- **warn_duplicate_name** (*bool*) – if `True` prints a warning when two different *Strand*'s have the same `VendorFields.name` and the same *Strand.dna_sequence*. An *IllegalDesignError* is raised (regardless of the value of this parameter) if two different *Strand*'s have the same name but different sequences, IDT scales, or IDT purifications.
- **only_strands_with_idt** (*bool*) – If False (the default), all non-scaffold sequences are output, with reasonable default values chosen if the field *Strand.vendor_fields* is missing. (though scaffold is included if *export_scaffold* is True). If True, then strands lacking the field *Strand.vendor_fields* will not be exported.
- **export_scaffold** (*bool*) – If False (the default), scaffold sequences are not exported. If True, scaffold sequences on strands output according to *only_strands_with_idt* (i.e., scaffolds will be exported, unless they lack the field *Strand.vendor_fields* and *only_strands_with_idt* is True).
- **export_non_modified_strand_version** (*bool*) – For any *Strand* with a *Modification*, also export a version of the *Strand* without any modifications. The name for this *Strand* is the original name with '_nomods' appended to it.

Return type
None

write_idt_plate_excel_file(**, directory='.', filename=None, key=None, warn_duplicate_name=False, only_strands_with_idt=False, export_scaffold=False, use_default_plates=True, warn_using_default_plates=True, plate_type=PlateType.wells96, export_non_modified_strand_version=False*)

Write `.xlsx` (Microsoft Excel) file encoding the strands of this *Design* with the field *Strand.vendor_fields*, suitable for uploading to IDT (Integrated DNA Technologies, Coralville, IA, https://www.idtdna.com/) to describe a 96-well or 384-well plate (https://www.idtdna.com/site/order/plate/index/dna/), with the output file having the same name as the running script but with `.py` changed to `.xlsx`, unless *filename* is explicitly specified. For instance, if the script is named `my_origami.py`, then the sequences will be written to `my_origami.xlsx`.

If the last plate as fewer than 24 strands for a 96-well plate, or fewer than 96 strands for a 384-well plate, then the last two plates are rebalanced to ensure that each plate has at least that number of strands, because IDT charges extra for a plate with too few strands: https://www.idtdna.com/pages/products/custom-dna-rna/dna-oligos/custom-dna-oligos

Parameters

- **directory** (*str*) – specifies a directory in which to place the file, either absolute or relative to the current working directory. Default is the current working directory.
- **filename** (*Optional[str]*) – custom filename if default (explained above) is not desired
- **key** (*Optional[Callable[[Strand], Any]]*) – key function used to determine order in which to output strand sequences. Some useful defaults are provided by *strand_order_key_function()*
- **warn_duplicate_name** (*bool*) – if `True` prints a warning when two different *Strand*'s have the same *Strand.name* and the same *Strand.dna_sequence*. An *IllegalDesignError* is raised (regardless of the value of this parameter) if two different *Strand*'s have the same name but different sequences, IDT scales, or IDT purifications.
- **only_strands_with_idt** (*bool*) – If False (the default), all non-scaffold sequences are output, with reasonable default values chosen if the field *Strand.vendor_fields* is missing. (though scaffold is included if *export_scaffold* is True). If True, then strands lacking the field *Strand.vendor_fields* will not be exported. If False, then *use_default_plates* must be True.

- **export_scaffold** (*bool*) – If False (the default), scaffold sequences are not exported. If True, scaffold sequences on strands output according to *only_strands_with_idt* (i.e., scaffolds will be exported, unless they lack the field *Strand.vendor_fields* and *only_strands_with_idt* is True).
- **use_default_plates** (*bool*) – Use default values for plate and well (ignoring those in idt fields, which may be None). If False, each Strand to export must have the field *Strand.vendor_fields*, so in particular the parameter *only_strands_with_idt* must be True.
- **warn_using_default_plates** (*bool*) – specifies whether, if *use_default_plates* is True, to print a warning for strands whose *Strand.vendor_fields* has the fields *VendorFields.plate* and *VendorFields.well*, since *use_default_plates* directs these fields to be ignored.
- **plate_type** (PlateType) – a *PlateType* specifying whether to use a 96-well plate or a 384-well plate if the *use_default_plates* parameter is `True`. Ignored if *use_default_plates* is `False`, because in that case the wells are explicitly set by the user, who is free to use coordinates for either plate type.
- **export_non_modified_strand_version** (*bool*) – For any *Strand* with a *Modification*, also export a version of the *Strand* without any modifications. The name for this *Strand* is the original name with '_nomods' appended to it.

Return type
None

write_oxview_file(*directory='.'*, *filename=None*, *warn_duplicate_strand_names=True*, *use_strand_colors=True*)

Writes an oxView file rerpesenting this design.

Parameters

- **directory** (*str*) – directy in which to write the file (default: current working directory)
- **filename** (*Optional[str]*) – name of the file to write (default: name of the running script with .oxview extension)
- **warn_duplicate_strand_names** (*bool*) – if True, prints a warning to the screen indicating when strands are found to have duplicate names. (default: True)
- **use_strand_colors** (*bool*) – if True (default), sets the color of each nucleotide in a strand in oxView to the color of the strand.

Return type
None

to_oxview_format(*warn_duplicate_strand_names=True*, *use_strand_colors=True*)

Exports to oxView format: https://github.com/sulcgroup/oxdna-viewer/blob/master/file-format.md

Parameters

- **warn_duplicate_strand_names** (*bool*) – if True, prints a warning to the screen indicating when strands are found to have duplicate names. (default: True)
- **use_strand_colors** (*bool*) – if True (default), sets the color of each nucleotide in a strand in oxView to the color of the strand.

Returns
string in oxView text format

Return type
str

to_oxview_json(*warn_duplicate_strand_names=True*, *use_strand_colors=True*)

Exports to oxView format: https://github.com/sulcgroup/oxdna-viewer/blob/master/file-format.md

Parameters

- **warn_duplicate_strand_names** (*bool*) – if True, prints a warning to the screen indicating when strands are found to have duplicate names. (default: True)
- **use_strand_colors** (*bool*) – if True (default), sets the color of each nucleotide in a strand in oxView to the color of the strand.

Returns
Python dict

Return type
dict

to_oxdna_format(*warn_duplicate_strand_names=True*)

Exports to oxdna format.

The three angles of each *HelixGroup* are interpreted to be applied in the following order: first *HelixGroup.yaw*, then *HelixGroup.pitch*, then *HelixGroup.roll*, using the "intrinsic rotation" convention (see https://en.wikipedia.org/wiki/Euler_angles#Conventions_by_intrinsic_rotations). The value *Helix.roll* is added to the value *HelixGroup.roll*.

Parameters
warn_duplicate_strand_names (*bool*) – If True, prints a warning to the screen indicating when strands are found to have duplicate names.

Returns
two strings that are the contents of the .dat and .top file suitable for reading by oxdna (https://sulcgroup.github.io/oxdna-viewer/)

Return type
Tuple[str, str]

write_oxdna_files(*directory='.'*, *filename_no_extension=None*, *warn_duplicate_strand_names=True*)

Write text file representing this *Design*, suitable for reading by oxdna (https://sulcgroup.github.io/oxdna-viewer/), with the output files having the same name as the running script but with `.py` changed to `.dat` and `.top`, unless *filename_no_extension* is explicitly specified. For instance, if the script is named `my_origami.py`, then the design will be written to `my_origami.dat` and `my_origami.top`.

The strings written are those returned by *Design.to_oxdna_format()*.

The three angles of each *HelixGroup* are interpreted to be applied in the following order: first *HelixGroup.yaw*, then *HelixGroup.pitch*, then *HelixGroup.roll*, using the "intrinsic rotation" convention (see https://en.wikipedia.org/wiki/Euler_angles#Conventions_by_intrinsic_rotations). The value *Helix.roll* is added to the value *HelixGroup.roll*.

Parameters

- **directory** (*str*) – directory in which to put file (default: current working directory)
- **filename_no_extension** (*Optional[str]*) – filename without extension (default: name of running script without `.py`).
- **warn_duplicate_strand_names** (*bool*) – If True, prints a warning to the screen indicating when strands are found to have duplicate names.

Return type
None

write_scadnano_file(*filename=None*, *directory='.'*, *extension=None*, *suppress_indent=True*, *warn_duplicate_strand_names=True*)

Write text file representing this *Design*, suitable for reading by the scadnano web interface, with the output file having the same name as the running script but with `.py` changed to *default_scadnano_file_extension*, unless *filename* is explicitly specified. For instance, if the script is named `my_origami.py`, then the design will be written to `my_origami.sc`. If *extension* is specified (but *filename* is not), then the design will be written to `my_origami.<extension>`

The string written is that returned by *Design.to_json()*.

Parameters

- **filename** (*Optional[str]*) – filename (default: name of script with `.py` replaced by `.sc`). Mutually exclusive with *extension*
- **directory** (*str*) – directory in which to put file (default: current working directory)
- **extension** (*Optional[str]*) – extension for filename (default: `.sc`) Mutually exclusive with *filename*
- **warn_duplicate_strand_names** (*bool*) – If True, prints a warning to the screen indicating when strands are found to have duplicate names.
- **suppress_indent** (*bool*) – whether to suppress indenting JSON for "small" objects such as short lists, e.g., grid coordinates. If True, something like this will be written:

```
{
  "grid_position": [1, 2]
}
```

instead of this:

```
{
  "grid_position": [
    1,
    2
  ]
}
```

Return type
None

write_cadnano_v2_file(*directory='.'*, *filename=None*, *whitespace=True*)

Write `.json` file representing this *Design*, suitable for reading by cadnano v2.

The string written is that returned by `Design.to_cadnano_v2()`.

If the cadnano file is intended to be used with CanDo (https://cando-dna-origami.org/), the optional parameter *whitespace* must be set to False.

Parameters

- **directory** (*str*) – directory in which to place the file, either absolute or relative to the current working directory. Default is the current working directory.
- **whitespace** (*bool*) – Whether to include whitespace in the exported file. Set to False to use this with CanDo (https://cando-dna-origami.org/), since that tool generates an error if the cadnano file contains whitespace.
- **filename** (*Optional[str]*) – The output file has the same name as the running script but with `.py` changed to `.json`, unless *filename* is explicitly specified. For instance, if

the script is named `my_origami.py`, then if filename is not specified, the design will be written to `my_origami.json`.

Return type
None

`add_nick`(*helix*, *offset*, *forward*, *new_color=True*)

Add nick to *Domain* on *Helix* with index *helix*, in direction given by *forward*, at offset *offset*. The two *Domain*'s created by this nick will have 5'/3' ends at offsets *offset* and *offset-1*.

For example, if there is a *Domain* with *Domain.helix* = `0`, *Domain.forward* = `True`, *Domain.start* = `0`, *Domain.end* = `10`, then calling `add_nick(helix=0, offset=5, forward=True)` will split it into two *Domain*'s, with one domains having the fields *Domain.helix* = `0`, *Domain.forward* = `True`, *Domain.start* = `0`, *Domain.end* = `5`, (recall that *Domain.end* is exclusive, meaning that the largest offset on this *Domain* is 4 = `offset-1`) and the other domain having the fields *Domain.helix* = `0`, *Domain.forward* = `True`, *Domain.start* = `5`, *Domain.end* = `10`.

If the *Strand* is circular, then it will be made linear with the 5' and 3' ends at the nick position, modified in place. Otherwise, this *Strand* will be deleted from the design, and two new *Strand*'s will be added.

Parameters

- **helix** (*int*) – index of helix where nick will occur
- **offset** (*int*) – offset to nick (nick will be between offset and offset-1)
- **forward** (*bool*) – forward or reverse *Domain* on *helix* at *offset*?
- **new_color** (*bool*) – whether to assign a new color to one of the *Strand*'s resulting from the nick. If False, both *Strand*'s created have the same color as the original. If True, one *Strand* keeps the same color as the original and the other is assigned a new color.

Return type
None

`ligate`(*helix*, *offset*, *forward*)

Reverse operation of *Design.add_nick()*. "Ligates" a nick between two adjacent *Domain*'s in the same direction on a *Helix* with index *helix*, in direction given by *forward*, at offset *offset*.

For example, if there are a *Domain*'s with *Domain.helix* = `0`, *Domain.forward* = `True`, *Domain.start* = `0`, *Domain.end* = `5`, (recall that *Domain.end* is exclusive, meaning that the largest offset on this *Domain* is 4 = `offset-1`) and the other domain having the fields *Domain.helix* = `0`, *Domain.forward* = `True`, *Domain.start* = `5`, *Domain.end* = `10`. then calling `ligate(helix=0, offset=5, forward=True)` will combine them into one *Domain*, having the fields *Domain.helix* = `0`, *Domain.forward* = `True`, *Domain.start* = `0`, *Domain.end* = `10`.

If the *Domain*'s are on the same *Strand* (i.e., they are the 5' and 3' ends of that *Strand*, which is necessarily linear), then the *Strand* is made is circular in place, Otherwise, the two *Strand*'s of each *Domain* will be joined into one, replacing the previous strand on the 5'-most side of the nick (i.e., the one whose 3' end terminated at the nick), and deleting the other strand.

Parameters

- **helix** (*int*) – index of helix where nick will be ligated
- **offset** (*int*) – offset to ligate (nick to ligate must be between offset and offset-1)
- **forward** (*bool*) – forward or reverse *Domain* on *helix* at *offset*?

Return type
None

add_half_crossover(*helix*, *helix2*, *offset*, *forward*, *offset2=None*, *forward2=None*)

Add a half crossover from helix *helix* at offset *offset* to *helix2*, on the strand with `Strand.forward` = *forward*.

Unlike *Design.add_full_crossover()*, which automatically adds a nick between the two half-crossovers, to call this method, there must *already* be nicks adjacent to the given offsets on the given helices. (either on the left or right side)

If the crossover is within a *Strand*, i.e., between its 5' and ' ends, the *Strand* will simply be made circular, modifying it in place. Otherwise, the old two *Strand*'s will be deleted, and a new *Strand* added.

Parameters

- **helix** (*int*) – index of one helix of half crossover
- **helix2** (*int*) – index of other helix of half crossover
- **offset** (*int*) – offset on *helix* at which to add half crossover
- **forward** (*bool*) – direction of *Strand* on *helix* to which to add half crossover
- **offset2** (*Optional[int]*) – offset on *helix2* at which to add half crossover. If not specified, defaults to *offset*
- **forward2** (*Optional[bool]*) – direction of *Strand* on *helix2* to which to add half crossover. If not specified, defaults to the negation of *forward*

Return type
None

add_full_crossover(*helix*, *helix2*, *offset*, *forward*, *offset2=None*, *forward2=None*)

Adds two half-crossovers, one at *offset* and another at *offset*-1. Other arguments have the same meaning as in *Design.add_half_crossover()*. A nick is automatically added on helix *helix* between *offset* and *offset*-1 if one is not already present, and similarly for *offset2* on helix *helix2*.

Parameters

- **helix** (*int*) – index of one helix of half crossover
- **helix2** (*int*) – index of other helix of half crossover
- **offset** (*int*) – offset on *helix* at which to add half crossover
- **forward** (*bool*) – direction of *Strand* on *helix* to which to add half crossover
- **offset2** (*Optional[int]*) – offset on *helix2* at which to add half crossover. If not specified, defaults to *offset*
- **forward2** (*Optional[bool]*) – direction of *Strand* on *helix2* to which to add half crossover. If not specified, defaults to the negation of *forward*

Return type
None

inline_deletions_insertions()

Converts deletions and insertions by "inlining" them. Insertions and deletions are removed, and their domains have their lengths altered. Also, major tick marks on the helices will be shifted to preserve their adjacency to bases already present. For example, if there are major tick marks at 0, 8, 18, 24, and a deletion between 0 and 8:

```
0       8          18    24    30
|--X---|----------|-----|------
```

then the domain is shortened by 1, the tick marks become 0, 7, 15, 23, and the helix's maximum offset is shrunk by 1:

```
0     7         17    23    29
|-----|---------|-----|------
```

Similarly, if there are insertions (in the example below, the "2" represents an insertion of length 2, which represents 3 total bases), they are expanded

```
0      8         18    24    30
|--2---|---------|-----|------
```

then the domain is lengthened by 3:

```
0        10        20    26    32
|--------|---------|-----|------
```

And it works if there are both insertions and deletions:

```
0      8         18    24    30
|--2---|---------|--X--|------
```

then the domain is lengthened by 3:

```
0        10        20   25    31
|--------|---------|----|------
```

We assume that a major tick mark appears just to the LEFT of the offset it encodes, so the minimum and maximum offsets for tick marks are respectively the helix's minimum offset and 1 plus its maximum offset, the latter being just to the right of the last offset on the helix.

Return type
None

`reverse_all()`

Reverses "polarity" of every *Strand* in this *Design*.

No attempt is made to make any assigned DNA sequences match by reversing or rearranging them. Every *Strand* keeps the same DNA sequence it had before (unreversed), if one was assigned. It is recommended to assign/reassign DNA sequences *after* doing this operation.

Return type
None

`strand_with_name(name)`

Parameters
name (*str*) – name of a *Strand*.

Returns
the *Strand* with name *name*, or None if no *Strand* in the *Design* has that name.

Return type
Optional[Strand]

`add_helix(idx, helix)`

Adds *helix* as a new *Helix* with index *idx* to this Design.

Parameters

- **idx** (*int*) – index of new *Helix*
- **helix** (Helix) – the new *Helix*

Return type
None

relax_helix_rolls()

Sets all helix rolls to "relax" them based on their crossovers.

This calculates the "strain" of each crossover c as the absolute value d_c of the distance between the angle to the helix to which it is connected and the angle of that crossover given the current helix roll. It minimizes sum_c d_c^2, i.e., minimize the sum of the squares of the strains.

Return type
None

scadnano.**write_file_same_name_as_running_python_script**(*contents*, *extension*, *directory='.'*, *filename=None*, *add_extension=False*)

Writes a text file with *contents* whose name is (by default) the same as the name of the currently running script, but with extension `.py` changed to *extension*.

Parameters

- **contents** (*str*) – contents of file to write
- **extension** (*str*) – extension to use
- **directory** (*str*) – directory in which to write file. If not specified, the current working directory is used.
- **add_extension** (*bool*) – whether to replace *.py* with *extension*
- **filename** (*Optional[str]*) – filename to use instead of the currently running script

Return type
None

scadnano.**grid_position_to_position**(*grid_position*, *grid*, *geometry*)

Converts a grid position to a 3D position (a *Position3D*).

Parameters

- **grid_position** (*Tuple[int, int]*) – pair of ints representing a grid position
- **grid** (Grid) – the *Grid*; cannot be *Grid.none*
- **geometry** (Geometry) – the *Geometry* object defining spacing parameters between grid positions

Returns
the *Position3D* represented by *grid_position* in the grid *grid*; Note that the *Position3D.z* coordinate is always 0.

Return type
Position3D

CHAPTER

TWO

ORIGAMI_RECTANGLE

The *origami_rectangle* module defines the function *origami_rectangle.create()* for creating a DNA origami rectangle using the *scadnano* module.

class origami_rectangle.**NickPattern**(*value*)

Represents options for where to place nicks between staples.

staggered = 1

A nick appears in a given helix and column if the parity of the helix and column match (both even or both odd).

staggered_opposite = 2

A nick appears in a given helix and column if the parity of the helix and column don't match (one is even and the other is odd).

CURRENTLY UNSUPPORTED.

even = 3

A nick appears in every column and only even-index helices.

CURRENTLY UNSUPPORTED.

odd = 4

A nick appears in every column and only odd-index helices.

CURRENTLY UNSUPPORTED.

origami_rectangle.**staggered = NickPattern.staggered**

Convenience reference defined so one can type *origami_rectangle.staggered* instead of *origami_rectangle.NickPattern.staggered*.

origami_rectangle.**staggered_opposite = NickPattern.staggered_opposite**

Convenience reference defined so one can type *origami_rectangle.staggered_opposite* instead of *origami_rectangle.NickPattern.staggered_opposite*.

CURRENTLY UNSUPPORTED.

origami_rectangle.**even = NickPattern.even**

Convenience reference defined so one can type *origami_rectangle.even* instead of *origami_rectangle.NickPattern.even*.

CURRENTLY UNSUPPORTED.

origami_rectangle.**odd = NickPattern.odd**

Convenience reference defined so one can type *origami_rectangle.odd* instead of *origami_rectangle.NickPattern.odd*.

CURRENTLY UNSUPPORTED.

origami_rectangle.**create**(*, *num_helices*, *num_cols*, *assign_seq=True*, *seam_left_column=-1*, *nick_pattern=NickPattern.staggered*, *twist_correction_deletion_spacing=0*, *twist_correction_start_col=1*, *twist_correction_deletion_offset=-1*, *num_flanking_columns=1*, *num_flanking_helices=0*, *custom_scaffold=None*, *edge_staples=True*, *scaffold_nick_offset=-1*)

Creates a DNA origami rectangle with a given number of helices and "columns" (16-base-wide region in each helix). The columns include the 16-base regions on the end where potential "edge staples" go, as well as the two-column-wide "seam" region in the middle.

Below is an example diagram of the staples created by this function.

Consider for example the function call `create(num_helices=8, num_cols=10, nick_pattern=origami_rectangle.staggered)`. The scaffold strand resulting from this call is shown below:

```
  #       C0       #       C1       #       C2       #       C3       #       C4
↪   #       C5       #       C6       #       C7       #       C8       #       C9
↪      #
H0 +---------------- ---------------- ---------------- ---------------- ----------
↪--- ---------------- ---------------- ---------------- ---------------- ----------
↪-----+
   |
↪
↪     |
H1 +---------------- ---------------- ---------------- ---------------- ----------
↪--+ +---------------- ---------------- ---------------- ---------------- ----------
↪-----+
↪  | |
H2 +---------------- ---------------- ---------------- ---------------- ----------
↪--+ +---------------- ---------------- ---------------- ---------------- ----------
↪-----+
   |
↪
↪     |
H3 +---------------- ---------------- ---------------- ---------------- ----------
↪--+ +---------------- ---------------- ---------------- ---------------- ----------
↪-----+
↪  | |
H4 +---------------- ---------------- ---------------- ---------------- ----------
↪--+ +---------------- ---------------- ---------------- ---------------- ----------
↪-----+
   |
↪
↪     |
H5 +---------------- ---------------- ---------------- ---------------- ----------
↪--+ +---------------- ---------------- ---------------- ---------------- ----------
↪-----+
↪  | |
H6 +---------------- ---------------- ---------------- ---------------- ----------
↪--+ +---------------- ---------------- ---------------- ---------------- ----------
↪-----+
```

(continues on next page)

(continued from previous page)

```
   |                                                                                                  ␣
↪                                                                                                   ␣
↪      |
H7 +---------------- ---------------- ---------------- ---------------- -------------
↪--] <---------------- ---------------- ---------------- ---------------- ----------
↪-----+
```

Helix indices are labelled `H0`, `H1`, … and column indices are labeled `C0`, `C1`, … Each single symbol `-`, `+`, `<`, `>`, `[`, `]`, `+` represents one DNA base, so each column is 16 bases wide. The `#` is a visual delimiter between columns and does not represent any bases, nor do spaces between the base-representing symbols. The 5' end of a strand is indicated with `[` or `]` and the 3' end is indicated with `>` or `<`. A crossover is indicated with

```
+
|
+
```

Below are the staples resulting from this same call.

```
  #       C0       #       C1       #       C2       #       C3       #       C4    ␣
↪    #       C5       #       C6       #       C7       #       C8       #       C9␣
↪      #
H0 <--------------+ +---------------- -------]<------+ +---------------- -------]<----
↪--+ +---------------- -------]<------+ +---------------- -------]<------+ +---------
↪-----]
                  | |                                 | |                                ␣
↪                                     | |                                 | |
H1 [--------------+ +------>[------+ +--------------+ +------>[------+ +------------
↪--+ +------>[------+ +--------------+ +------>[------+ +--------------+ +---------
↪----->
                                    | |                                 | |           ␣
↪                   | |                                 | |
H2 <--------------+ +--------------+ +------]<------+ +--------------+ +------]<----
↪--+ +--------------+ +------]<------+ +--------------+ +------]<------+ +---------
↪-----]
                  | |                                 | |                              ␣
↪                                     | |                                 | |
H3 [--------------+ +------>[------+ +--------------+ +------>[------+ +------------
↪--+ +------>[------+ +--------------+ +------>[------+ +--------------+ +---------
↪----->
                                    | |                                 | |           ␣
↪                   | |                                 | |
H4 <--------------+ +--------------+ +------]<------+ +--------------+ +------]<----
↪--+ +--------------+ +------]<------+ +--------------+ +------]<------+ +---------
↪-----]
                  | |                                 | |                              ␣
↪                                     | |                                 | |
H5 [--------------+ +------>[------+ +--------------+ +------>[------+ +------------
↪--+ +------>[------+ +--------------+ +------>[------+ +--------------+ +---------
↪----->
                                    | |                                 | |           ␣
↪                   | |                                 | |
H6 <--------------+ +--------------+ +------]<------+ +--------------+ +------]<----
```

(continues on next page)

(continued from previous page)

```
↪--+ +--------------+ +------]<------+ +--------------+ +------]<------+ +---------
↪-----]
                  | |                                | |                          ␣
↪                                    | |                                | |
H7 [--------------+ +------>[------- ---------------+ +------>[------- -------------
↪--+ +------>[------- ---------------+ +------>[------- ---------------+ +---------
↪----->
```

The seam crosses columns `C4` and `C5`. The left and right edge staples respectively are in columns `C0` and `C9`.

Prints warning if number of bases exceeds 7249 (length of standard M13 scaffold strand), but does not otherwise cause an error.

Here's an example of using *origami_rectangle.create* to create a design for a 16-helix rectangle and write it to a file readable by scadnano. (By default the output file name is the same as the script calling *scadnano.Design.write_scadnano_file()* but with the extension `.py` changed to `.dna`.)

```
import origami_rectangle as rect

# XXX: ensure num_cols is even since we divide it by 2
design = rect.create(num_helices=16, num_cols=24, nick_pattern=rect.staggered)
design.write_scadnano_file()
```

However, we caution that *create()* is not intended to be very extensible for creating many different types of DNA origami. It is more intended as an example whose source code can be an efficient reference to learn the *scadnano* API.

Parameters

- **num_helices** (*int*) – number of helices. must be even.
- **num_cols** (*int*) – number of "columns" as defined above. must be even and at least 4.
- **assign_seq** (*bool*) – whether to assign a DNA sequence to the scaffold. If True, uses *custom_scaffold* if it is not None, or M13 otherwise.
- **seam_left_column** (*int*) – specifies the location of the seam. (i.e., scaffold crossovers in the middle of the origami.) If positive, the seam occupies two columns, and *seam_left_column* specifies the column on the left. To make the crossover geometry work out, a nonnegative *seam_left_column* must be even, greater than 0, and less than *num_helices* - 2. If negative, it is calculated automatically to be roughly in the middle.
- **nick_pattern** (NickPattern) – describes whether nicks between staples should be "staggered" or not. See *origami_rectangle.NickPattern* for details.
- **twist_correction_deletion_spacing** (*int*) – If *twist_correction_deletion_spacing* > 0, adds deletions between crossovers in one out of every *twist_correction_deletion_spacing* columns. See this paper for an explanation of twist correction in DNA origami: *Programmable molecular recognition based on the geometry of DNA nanostructures*, Sungwook Woo and Paul W. K. Rothemund, Nature Chemistry, volume 3, pages 620–627 (2011) https://doi.org/10.1038/nchem.1070
- **twist_correction_start_col** (*int*) – ignored if *twist_correction_deletion_spacing* <= 0, otherwise it indicates the column at which to put the first deletions. Default = 1.
- **twist_correction_deletion_offset** (*int*) – the *relative* offset of the deletion, relative to the left side of the column.

- **num_flanking_columns** (*int*) – the number of empty columns on the helix on each side of the origami.
- **num_flanking_helices** (*int*) – the number of empty helices above and below the origami.
- **custom_scaffold** (*Optional[str]*) – the scaffold sequence to use. If set to None, the standard 7249-base M13 is used: *scadnano.m13()*.
- **edge_staples** (*bool*) – indicates whether to include the edge staples. (Leaving them out prevents multiple origami rectangles from polymerizing in solution due to base stacking interactions on the left and right edges of the origami rectangle.)
- **scaffold_nick_offset** (*int*) – the position of the "nick" on the scaffold (the M13 scaffold is circular, so for such a scaffold this really represents where any unused and undepicted bases of the scaffold will form a loop-out). If negative (default value) then it will be chosen to be along the origami seam.

Returns
a *Design* representing a DNA origami rectangle

Return type
Design

CHAPTER

THREE

INTEROPERABILITY - CADNANO V2

Scadnano provides function to convert design to and from cadnano v2:

- `scadnano.DNADesign.from_cadnano_v2()` will create a scadnano DNADesign from a `cadnanov2` json file.
- `scadnano.DNADesign.export_cadnano_v2()` will produce a `cadnanov2` json file from a scadnano design.

Important

All `cadnanov2` designs can be imported to scadnano. However **not all scadnano designs can be imported to cadnanov2**, to be importable to `cadnanov2` a scadnano design need to comply with the following points:

- The design cannot feature any `Loopout` as it is not a concept that exists in `cadnanov2`.
- Following `cadnanov2` conventions, helices with **even** number must have their scaffold going **forward** and helices with **odd** number **backward**.

Also note that maximum helices offsets can be altered in a `scadnano` to `cadnanov2` conversion as `cadnanov2` needs max offsets to be a multiple of 21 in the hex grid and 32 in the rectangular grid. The conversion algorithm will choose the lowest multiple of 21 or 32 which fits the entire design.

The `cadnanov2` json format does not embed sequences hence they will be lost after conversion.

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z