Sequences!

What is a Sequence?

- An **Abstract Data Type** that is an ordered, 0-indexed set of values.

- There are many specific *types* of sequences with their own properties. Common, built-in sequence types in Python include:

1. str - a sequence of character data
2. Tuple - a fixed-size sequence of values of any types
3. List - a dynamically-sized sequence of values of a specific type
4. range - a sequence of integers at intervals between a start and end

Tuples!

Tuple Types

1. Import the type definition for List from the standard **`typing`** library

```
from typing import Tuple
```

2. Tuples types are *made of a specific, fixed-length sequence of any mixed type(s)* by:

```
Tuple[type_0, type_1, ..., type_N]
```

3. Typically you will want to alias your Tuple types to give them a more meaningful name

Examples:

```
Point2D = Tuple[float, float]
Color = Tuple[int, int, int]
```

4. You **construct** a Tuple with a Tuple literal. Tuple variables of the above types could be initialized as follows:

```
origin: Point2D = (0.0, 0.0)
gray: Color = (128, 128, 128)
```

Lists !

Lists are a sequence of values of the same type... ...and can change at runtime!

a: List[int]	int	int	int	int	int	int	int	int
index:	0	1	2	3	4	5	6	7

1. Each item in a List* is called an ***item*** or an ***element***

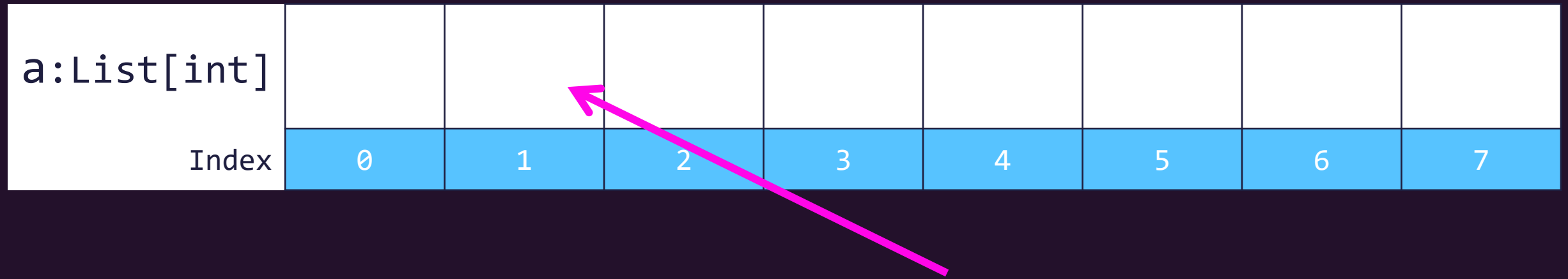
2. An element is a single value **addressed by its index** ("Room #")

3. All elements in a List are of the **same *type*** **
   - An array of `ints, floats, strings, bools, and so on.`

* *Other languages may use the term array instead of list and may have subtly different characteristics.*

** *Technically*, in Python, you can create lists where elements are of many different types. While this flexibility sounds nice, the unpredictability of it is difficult to reason about in practice and is a common source of accidental errors. It is generally advised for lists to work with a *single type of data.*

Elements are addressed by the array variable's **name** and **index**



1. Notation: **`array_name[index]`**, i.e. **`a[1]`**

2. **Indexing starts at [0]** (not [1])
   - First index *always* 0
   - Last index *always* length of array – 1
   - This is a convention shared by most programming languages

Declaring and Initializing Lists

1. Import the type definition for List from the standard **`typing`** library*

```
from typing import List
```

2. You can **declare a List** of *any type* by

`<identifier>: List[type];` – list of <type>
`ages: List[int]` – list of int values
`words: List[str]` – list of str values

3. You **construct** an empty list in two ways:
   1. Use the List constructor with no argument: `list()`
   2. Use List literal with no elements: `[ ]`

4. These two initialization tasks are often done at the same time:

```
words: List[str] = []
```

List Literals

- Initializing a List with a sequence of elements is frequently useful
- Using List Literal syntax, you can do this directly:

```
ages: List[int] = [18, 21, 20, 18, 19, 19]
words: List[str] = ["the", "quick", "brown", "fox", "jumped"]
```

- The List Literal syntax is a sequence of expressions, separated by commas, whose types match the List's type.
- There are other ways to initialize non-empty Lists you'll soon learn!
  1. Iterator-based initialization
  2. List comprehensions

Appending Elements to a List

- Lists are a *mutable* data structure that can grow (or shrink) in length!
  - Unlike Tuples and Strings!

- The **append** method adds an element to the end of a List
  - The element to add is the method's only parameter
  - The method returns None, because it *mutates* the List

- Examples:

```
ages.append(22)
words.append("over")
```

Removing Elements from a List

- The **pop** method removes an element at a given index from a List
  - The **index** to remove is the method's only parameter
  - The method returns the value previously stored at that index

- If no index is provided, the pop method defaults to the last index

- If the popped index is in the middle of the list, the indices of all following elements move back by one to avoid a "gap" in the middle of a list.

- Example:

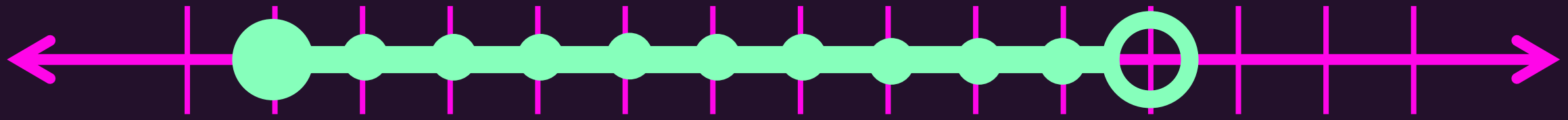
```
ages: List[int] = [18, 19, 20, 21]

print(ages.pop(1))   # 19
print(ages)          # [18, 20, 21]
print(ages.pop())    # 21
print(ages)          # [18, 20]
```

Fundamental List Operations

Operation	Form	Example
Declaration	`name: List[type]`	`scores: List[int]`
Construction (Empty)	`name = []`	`scores = []`
Construction (Non-empty)	`name = [<comma separated values>]`	`scores = [12, 0, 9]`
# of Elements	`len(name)`	`len(scores)`
Access Element	`name[index]`	`scores[0]`
Assign Element	`name[index] = expression`	`scores[1] = 12`
Append Element Returns None.	`name.append(expression)`	`scores.append(13)`
Remove Element Returns removed element.	`name.pop(index_expression)`	`scores.pop(1)`

Ranges !

Ranges of Integers



- What are the ***attributes*** of the ***range*** above?

- A **start** point that is inclusive
- A **stop** point that is exclusive
- A **step** that moves up by one

The `range` type *models* the *idea* of a Range

- `range` is a built-in *sequence type* in Python
  - Just like `str`, `Tuple`, and `List`
  - A range value is immutable, like `str` and `Tuple`
  - Documentation: https://docs.python.org/3/library/stdtypes.html#ranges

- The `range` constructor returns a range object

```
range(start: int, stop: int[, step: int = 1]) -> range
```

  - **start** is *inclusive.*
  - **stop** is exclusive
  - **step** defaults to **1** and is *optional,* as denoted by the brackets

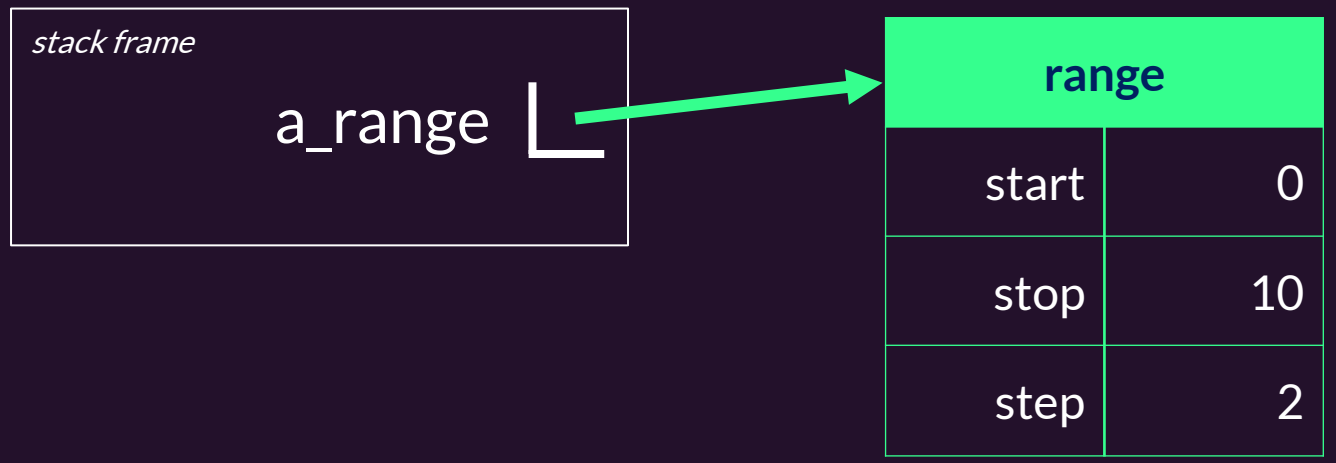
A `range` object has *attributes*

- ***Attributes*** are named values bundled in an object
  - *Attributes* represent the *state* of an object
  - **Named** like variables, unlike indexed items of a tuple or list. Attribute names are *identifiers.*
  - Hold **Values**, also like variables, unlike *methods* which are special functions

- Attributes are accessed using the dot operator following the object:
  `[object].[attribute_name]`

- Example:

```
>>> a_range = range(0, 10, 2)
>>> a_range.start
0
>>> a_range.stop
10
>>> a_range.step
2
```



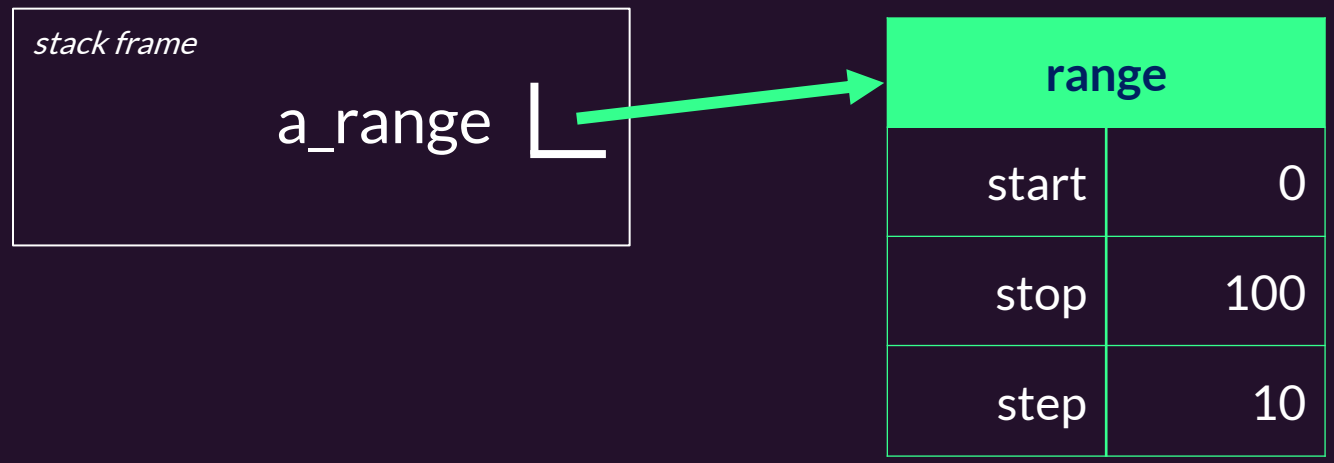
- The range object's attributes are read-only, making a range an *immutable object*

A `range` object is a *sequence* type

- You can access items in a range's sequence *by its index* using subscription:
  - range[0], range[1], ..., range[N]

- Example:

```
>>> a_range = range(0, 100, 10)
>>> a_range[0]
0
>>> a_range[1]
10
>>> a_range[9]
90
>>> a_range[10]
IndexError: range object index out of range
```



- Notice the *range* object's state is **only** its three attributes
  - *But* as a *sequence type*, with subscription, it also behaves as if it is made of many more items.
  - How? **Abstraction**! In this case the **abstraction** of a range is fully **represented** by just three attributes.

- This abstraction is possible through arithmetic
  `range[index]` evaluates to `range.start + (range.step * index)`