Python

A quickstart into the key concepts of programming

Data structures

Key concepts in programming

- Variables (*integers, strings, dates, etc.)*
- Flow control (*if then, loop, etc.*)
- Functions (*list of steps the code will follow*)

Built-in collection data types

Data structures

- Four built-in data structures (container): **list**, **tuple**, **dictionary**, and **set**.
- Can contain objects of *any* type
- Organize the data structure into four different families:
  - **Ordered data structure**: list and tuple
  - **Unordered data structure**: set and dictionary
  - **Mutable**: list, set and dictionary
  - **Immutable**: tuple
- A *mutable* data structure can change its size, whereas an *immutable* data structure will always maintain the same size.

Data types

Type Name	Example	Description	Notebook
list	[1, 2, 3]	Ordered collection	*list.ipynb*
tuple	(1, 2, 3)	Immutable ordered collection	*tuple.ipynb*
dict	{'a':1, 'b':2, 'c':3}	Unordered (key:value) pair mapping	*dict.ipynb*
set	{1, 2, 3}	Unordered collection of unique values	*set.ipynb*

Operations on Any Sequence in Python

Operation Name	Operator	Explanation
indexing	[ ]	Access an element of a sequence
concatenation	+	Combine sequences together
repetition	*	Concatenate a repeated number of times
membership	in	Ask whether an item is in a sequence
length	len	Ask the number of items in the sequence
slicing	[:]	Extract a part of a sequence

- https://runestone.academy/runestone/books/published/pythonds/Introduction/GettingStartedwithData.html

list

list.ipynb

Lists

- Holds an ordered collection of items, a sequence of values
- Lists are ordered collections of objects.
  - The elements of a list don't have to be the same type. Each element of the list can be of any type, including another list.
  - Elements can be referenced by an *index*.
- indexing: list in Python starts from 0!
- Create the list with brackets [ ].
  - Inside the brackets, the elements are separated by a comma (,).
  - `test_list = ['test1','test2',3]`

List indexing and slicing

- The syntax for accessing the elements of a list is the same as for accessing the characters of a string: the bracket operator. The expression inside the brackets specifies the index
- *Indexing:* fetching a single value from the list.

```
L = [2, 3, 5, 7, 11]
```

- Python uses *zero-based* indexing

0	1	2	3	positive index
A	B	C	D	sequence elements
-4	-3	-2	-1	negative index

```
L[0]
2
L[1]
3
```

- Elements at the end of the list can be accessed with negative numbers, starting from -1:

```
L[-1]
11
```

List indexing and slicing

- *Slicing:* accessing multiple values in sublists.
  It uses a colon to indicate the start point (inclusive) and end point (noninclusive) of the subarray

```
L[0:3]
[2, 3, 5]
```

- `a[start:end] # items start through end-1`
- `a[start:]    # items start through the rest of the array`
- `a[:end]      # items from the beginning through end-1`
- `a[:]         # a copy of the whole array`
- `a[start:end:step] # start through not past end, by step`
  - Note `:end` value represents the first value that is not in the selected slice

Enlarging Lists

- Initialize an empty list

```
L1=[]
```

- Initialize a list of known size, all elements to same value

```
L1=[0]*N
```

- Append

```
L1.append("Graham")
```

  - append adds its argument as a single element to the end of a list. The length of the list itself will increase by one.

- Extend

```
L1.extend(["Graham","Michael"])
```

  - extend iterates over its argument adding each element to the list, extending the list. The length of the list will increase by the number of elements in the iterable argument.
  - needs a list as argument

Enlarging Lists

- Concatenate

```
L=[1,2,3]+[4,5,6]
```

- Insert

```
L.insert(i,item)
```

  - insert item *before* element i
  - add an item at the beginning of a list, use `L.insert(0,item)`

Shorten Lists

- Delete an element by its index, remove an element or slice from a list

```
del L[i]
```

- Remove the first instance of a value in a list

```
L.remove(item)
```

  - The item must match exactly or an error occurs.
- Remove and return an element

```
Lastval=L.pop()
A_val=L.pop(2)
```

  - If no element is specified, the last element is returned. If it is present that element is returned.

More Operators, Methods on Lists

- length of a list

```
LenOfL=len(L)
```

- Maximum or minimum value of the items:

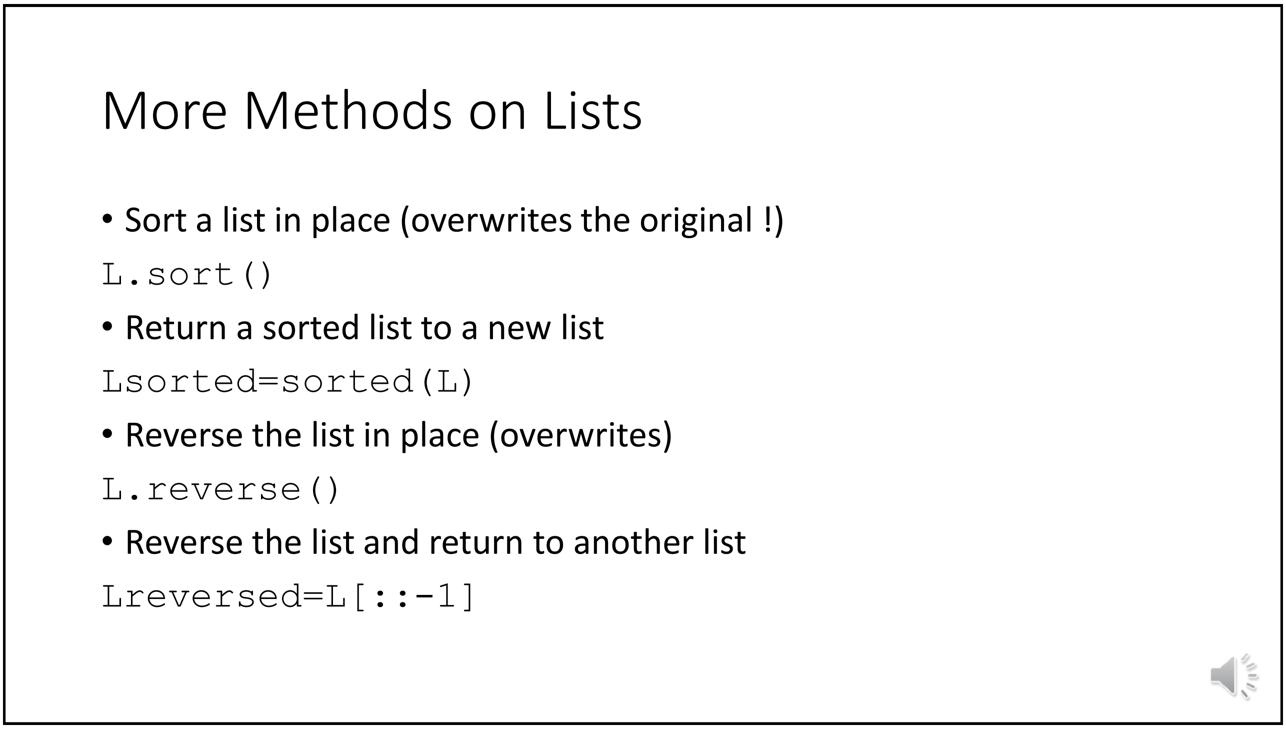
```
max(L), min(L)
```

- Index of first time item occurs

```
myIndex=L.index(item)
```

- Number of times item occurs

```
NumItem=L.count(item)
```



More Methods on Lists

- Sort a list in place (overwrites the original !)

```
L.sort()
```

- Return a sorted list to a new list

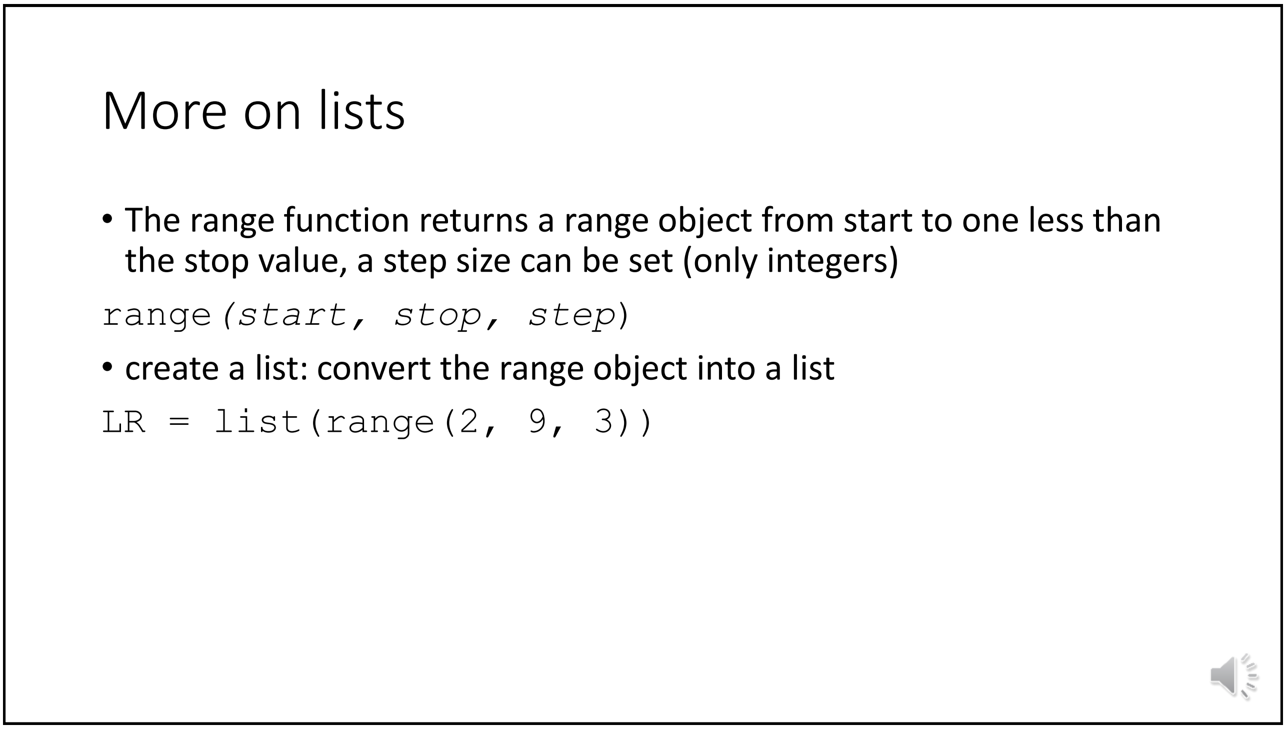
```
Lsorted=sorted(L)
```

- Reverse the list in place (overwrites)

```
L.reverse()
```

- Reverse the list and return to another list

```
Lreversed=L[::-1]
```



More on lists

- The range function returns a range object from start to one less than the stop value, a step size can be set (only integers)

```
range(start, stop, step)
```

- create a list: convert the range object into a list

```
LR = list(range(2, 9, 3))
```

More on lists

- The range function returns a list of numbers that range from zero to one less than the parameter

```
range(start, stop, step)
```

- Nested lists
  - Lists can hold any objects
  - File: list_nested.py
- Watch out for aliasing / copying lists
  - File:list_copy.py

Mutable Objects and values: aliasing

- If a refers to an object and you assign b = a, then both variables refer to the same object:

```
a = [1, 2, 3]
b = a
b is a
True
```

- The association of a variable with an object is called a reference.
- If the aliased object is mutable, changes made with one alias affect the other:

```
b[0] = 17
print(a)
[17, 2, 3]
```

https://eng.libretexts.org/Bookshelves/Computer_Science/Book:_Python_for_Everybody_(Severance)

Shallow / deep copy

- Although this behavior can be useful, it is **error-prone**. In general, it **is safer to avoid aliasing when you are working with mutable objects.**
- Instead of aliasing lists, make a copy
  - `c = a[:] # use a slice`
  - `c = a.copy()`
  - Python’s standard library provides the copy module, which provides copy functions that can be used to create copies of objects.
- For immutable objects like strings, aliasing is not as much of a problem.
- *File:list_copy.py*
- https://realpython.com/copying-python-objects/

tuple

tuple.ipynb

Tuples

- Tuples are defined by specifying items separated by commas within an optional pair of parentheses ().
- Because tuples are **immutable** objects, they are usually used when the list of values doesn't change.
  - `test_tuple_new = ('element1','element2','element3')`
- Note: Mutable *elements* of a tuple can be changed.

Tuple operations

- Comma-separated lists with no enclosing parentheses/brackets/braces are assumed tuples

```
T=1,2,3
type(T)
tuple
```

- create a tuple with a single element, include the final comma

```
T1 = (11,)
```

- Indexing

```
print T[0]
```

- Slicing

```
T2=T[1:]
```

- File: tuple_1.py

Tuple operations

- Length

```
len(T)
```

- Concatenation (note assignment to new variable)

```
T3=T+T2
```

- Membership

```
3 in T
```

- Iteration

```
for i in T:
    print i
```

Tip: Tuple or List?

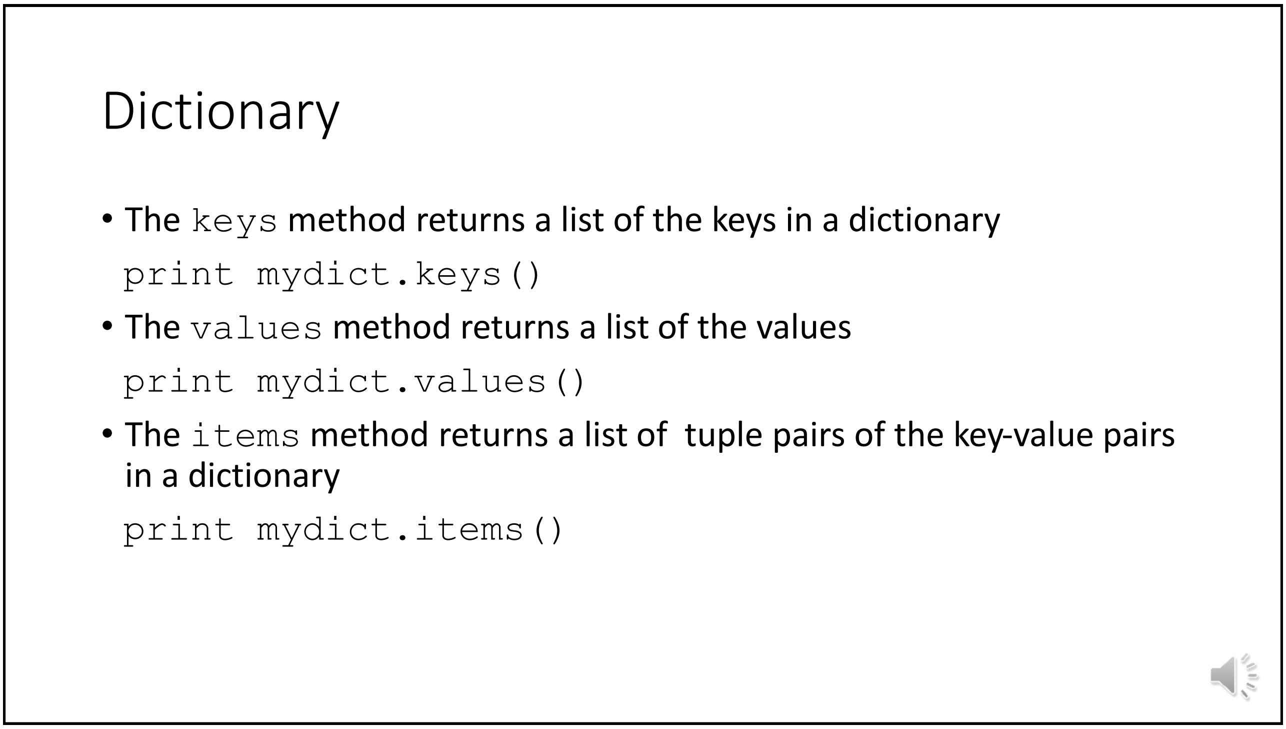
- Tuples are frequently used to return multiple variables from *functions*.
- Tuples should be used whenever the structure should not be dynamically sized or changed.
- Generally,
  - lists are for looping; tuples for structs.
  - Lists are homogeneous; tuples heterogeneous.
  - Lists for variable length. (https://stackoverflow.com/questions/16940293/why-is-there-no-tuple-comprehension-in-python)

dictionary

dictionary.ipynb

Dictionary

- A dictionary is like a list, but more general.
  - In a list, the index positions have to be integers;
  - in a dictionary, the indices can be (almost) any type.
- Mapping between a set of indices (keys) and a set of values. Each key maps to a value: a **key:value** pair structure, and it's possible to retrieve the value using the key.
- The key of the dictionary can be only created by using an immutable object, and the value can be either a mutable or immutable object.
- Key-value pairs in a dictionary are created using the notation
  - `mydict = {key : value, key : value},`
  - `mydict = {'ab' : 'abcd','cd' : 'efgh'}`
- To access a member of the dictionary, use the following syntax:
  - `mydict['ab']`



Dictionary

- The `keys` method returns a list of the keys in a dictionary

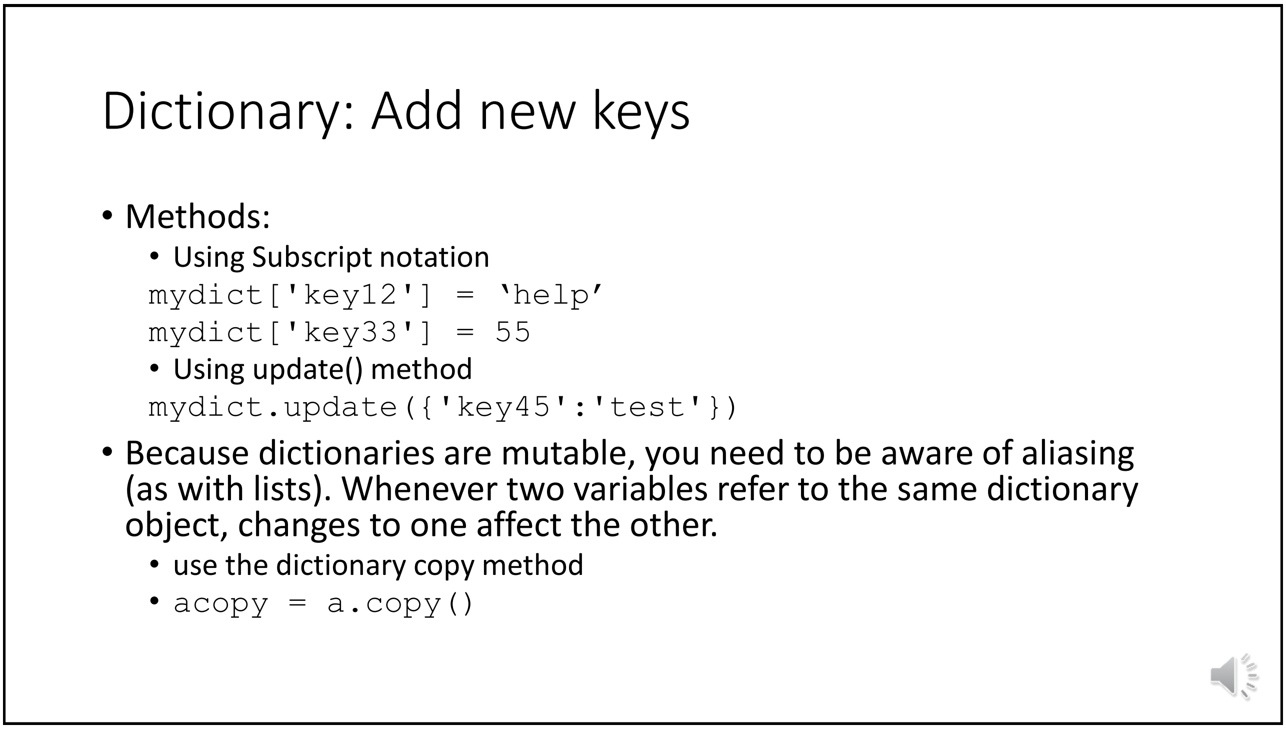
```
print mydict.keys()
```

- The `values` method returns a list of the values

```
print mydict.values()
```

- The `items` method returns a list of tuple pairs of the key-value pairs in a dictionary

```
print mydict.items()
```



Dictionary: Add new keys

- Methods:
  - Using Subscript notation

```
mydict['key12'] = ‘help’
mydict['key33'] = 55
```

  - Using update() method

```
mydict.update({'key45':'test'})
```

- Because dictionaries are mutable, you need to be aware of aliasing (as with lists). Whenever two variables refer to the same dictionary object, changes to one affect the other.
  - use the dictionary copy method
  - `acopy = a.copy()`

Dictionary

- Lists can be sorted with the sort() function
- dictionaries cannot be sorted, they are in no particular order

set

set.ipynb

Sets

- Sets are unordered collections of unique simple elements.
- Set elements are unique. Duplicate elements are not allowed

-
  ```
  S1={1, 2, 2, 2, 3, 4, 4, 5, 5}
  >>> S1
  {1, 2, 3, 4, 5}
  ```
- A set itself may be modified, but the elements contained in the set must be of an immutable type.

Sets

- Python has an in-built function `set()`, using which a set object can be constructed out of any sequence such as a string, list or a tuple object.
- `s1=set('help')`
- `s1`
- `{'e', 'h', 'l', 'p'}`
- `s2=set([5, 7,87, 55, 100])`
- `s2`
- `{5, 7, 55, 87, 100}`
- `s3=set((10,'go',15.3))`
- `s3`
- `{10, 15.3, 'go'}`

Source: https://www.tutorialsteacher.com/python/python-set

Sets

- A set is used when the collection is more important than the order of the elements or how many times they occur:

```
primes = {2, 3, 5, 7}
odds = {1,3,5,7,9}
```

- Cfr mathematical sets: union, intersection, difference

```
primes.union(odds)
primes.intersection(odds)
primes.difference(odds)
```

Sets: built-in methods

- `add()`
  - Adds a new element in the set object.
- `update()`
  - Adds multiple items from a list or a tuple.
- `clear()`
  - Removes the contents of set object and results in an empty set.
- `copy()`
  - Creates a copy of the set object.
- `discard()`
  - Returns a set after removing an item from it. No changes are done if the item is not present.
- `remove()`
  - Returns a set after removing an item from it. Results in an error if the item is not present.

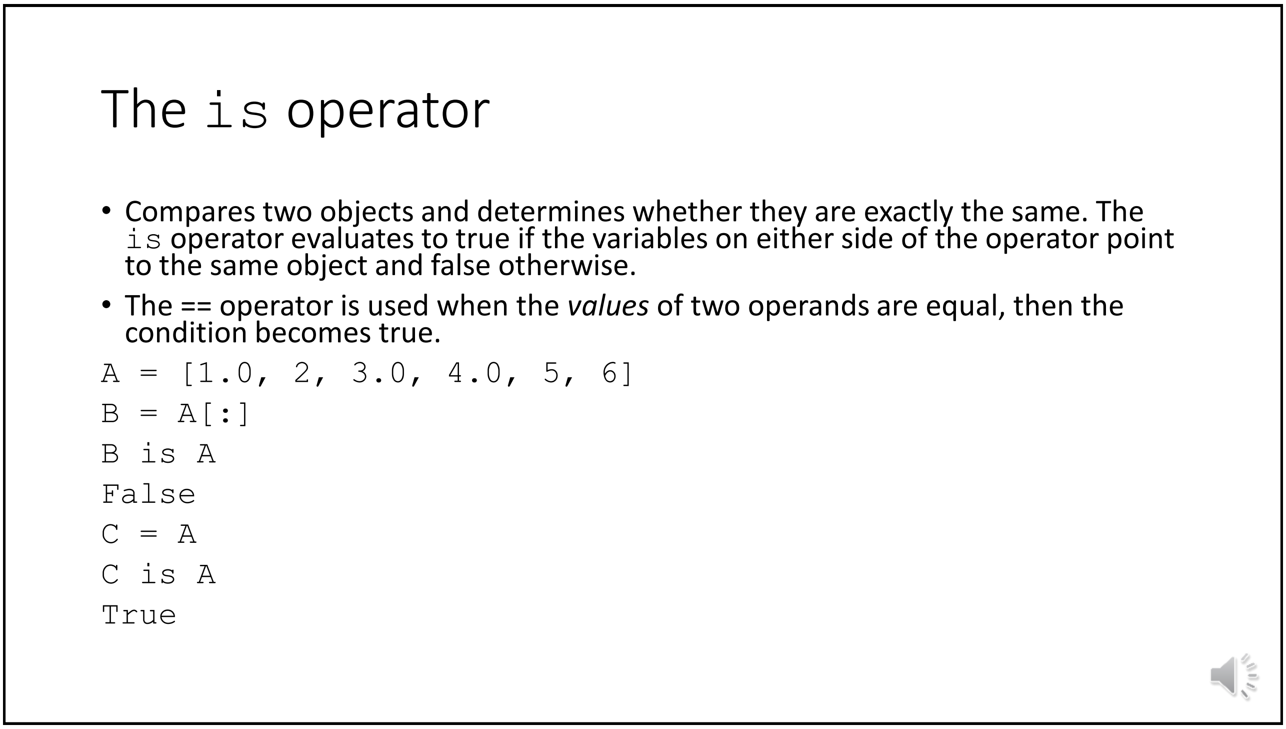
extra

The `in` operator

- Determines whether an item is a member of a sequence.
  - Sequence: anything ordered: string, list, tuples, etc.
- Returns: True or False

```
A=[1.,2,3.,4.,5,6]
[1.0, 2, 3.0, 4.0, 5, 6]
1 in A
True
1.0 in A
True
7 in A
False
```

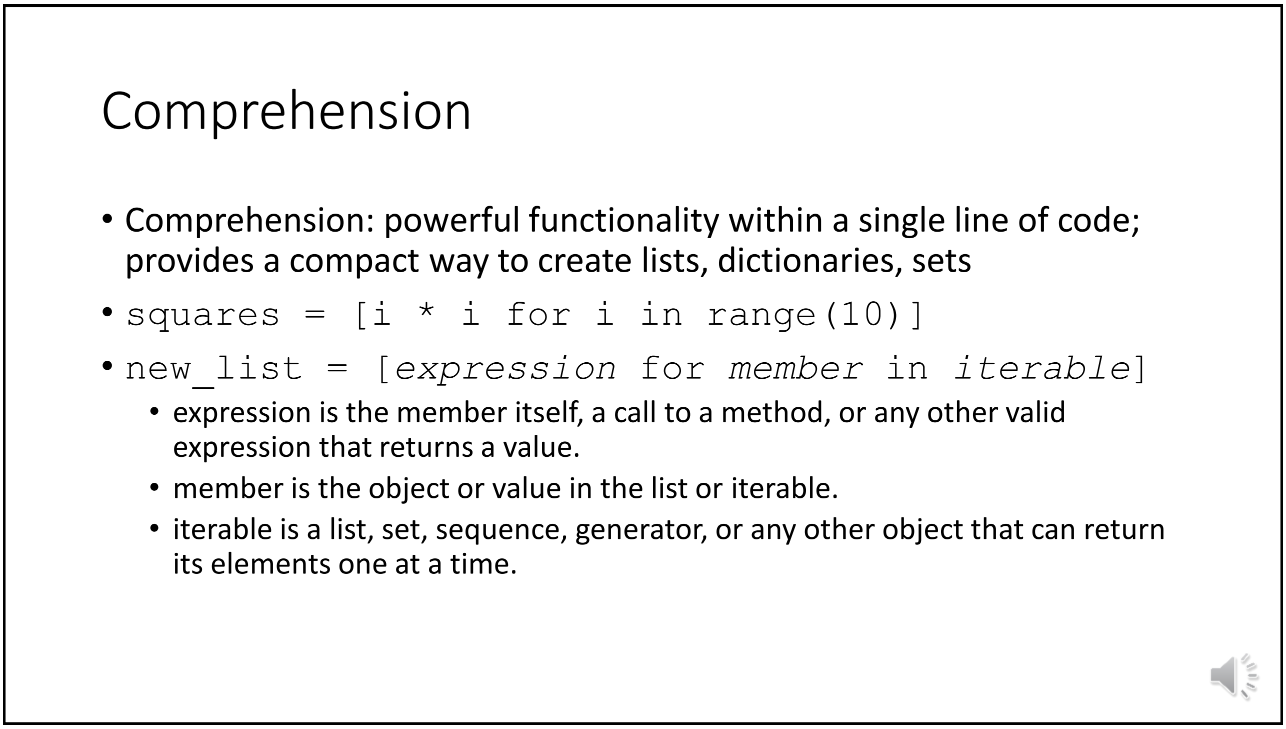
- Negation: `not in`



The `is` operator

- Compares two objects and determines whether they are exactly the same. The `is` operator evaluates to true if the variables on either side of the operator point to the same object and false otherwise.
- The == operator is used when the *values* of two operands are equal, then the condition becomes true.

```
A = [1.0, 2, 3.0, 4.0, 5, 6]
B = A[:]
B is A
False
C = A
C is A
True
```



Comprehension

- Comprehension: powerful functionality within a single line of code; provides a compact way to create lists, dictionaries, sets
- `squares = [i * i for i in range(10)]`
- `new_list = [`*`expression`*` for `*`member`*` in `*`iterable`*`]`
  - expression is the member itself, a call to a method, or any other valid expression that returns a value.
  - member is the object or value in the list or iterable.
  - iterable is a list, set, sequence, generator, or any other object that can return its elements one at a time.