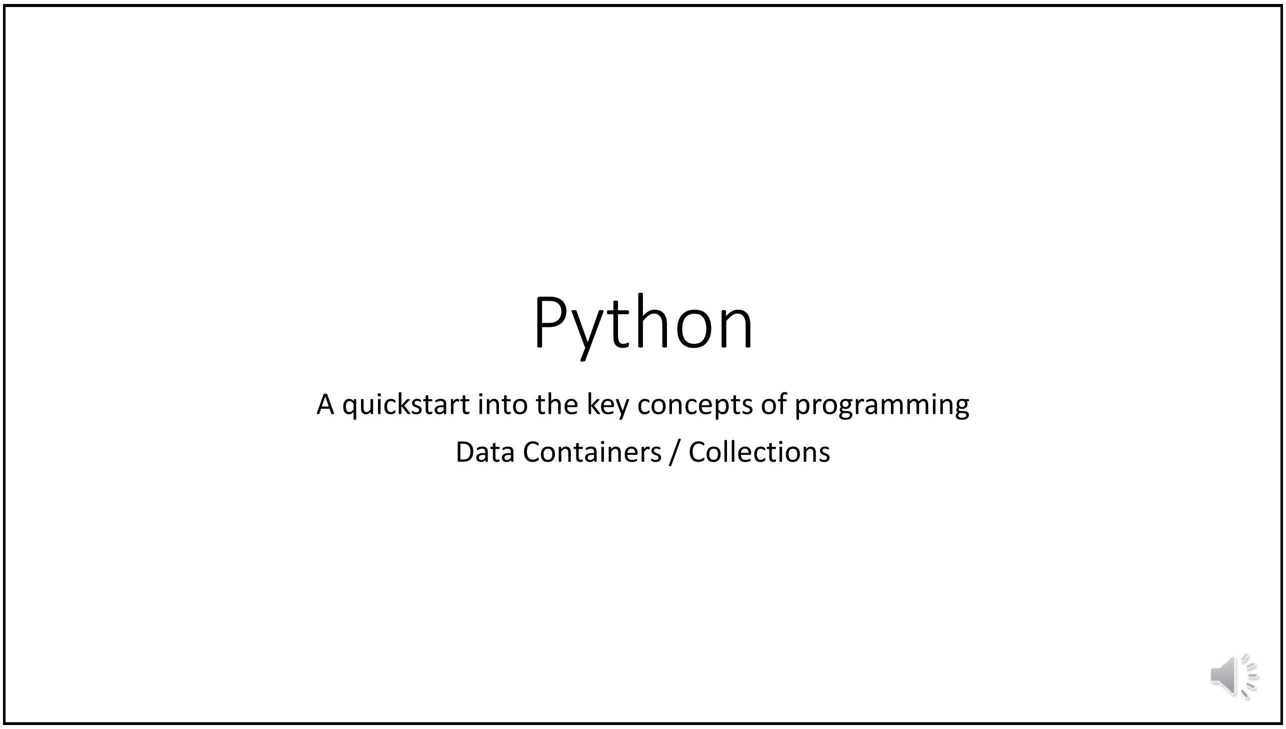
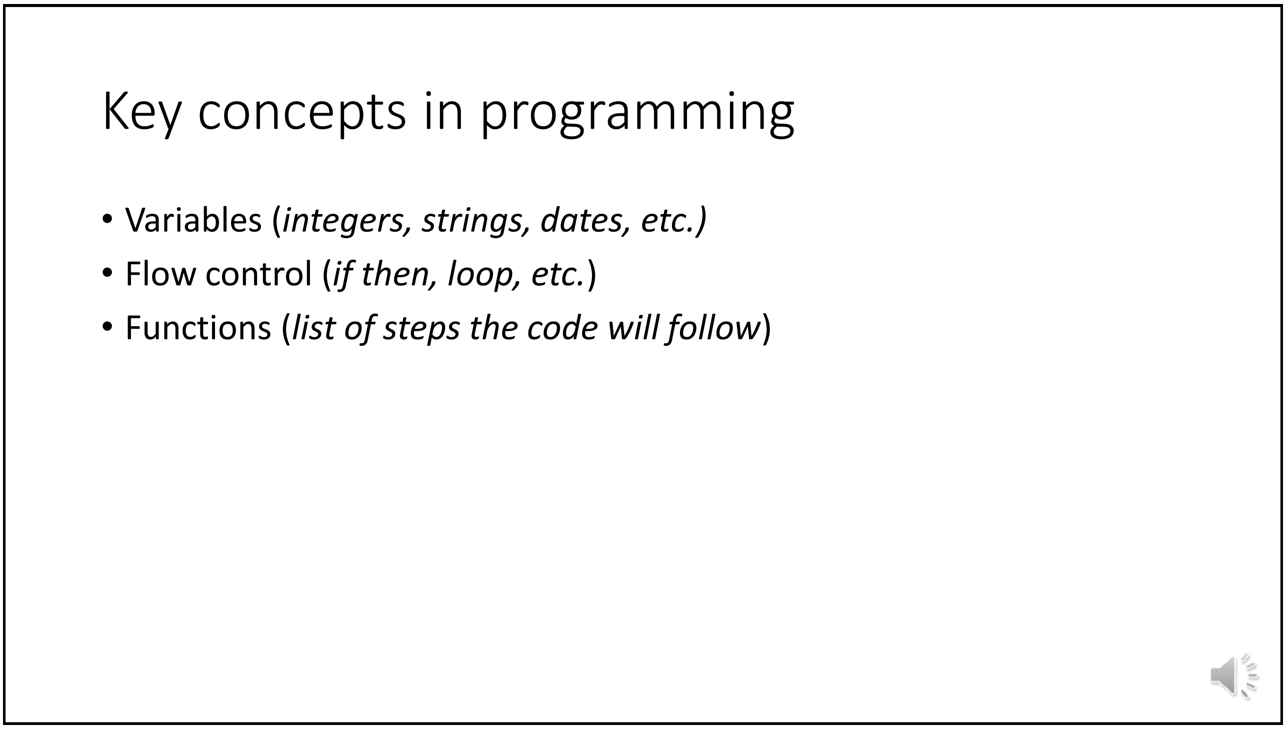


Python

A quickstart into the key concepts of programming

Data Containers / Collections



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

Compound data types that contain multiple objects

A.k.a. collections, containers

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 structures

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!

List creation

- Create the list with brackets [ ].
  - Inside the brackets, the elements are separated by a comma (,).

```
l1 = [1, 2, 3]

l2 = ['test1', 'test2’, 3*9]

l3 = ['help', [1, 2], 1, 2]
```

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]
```

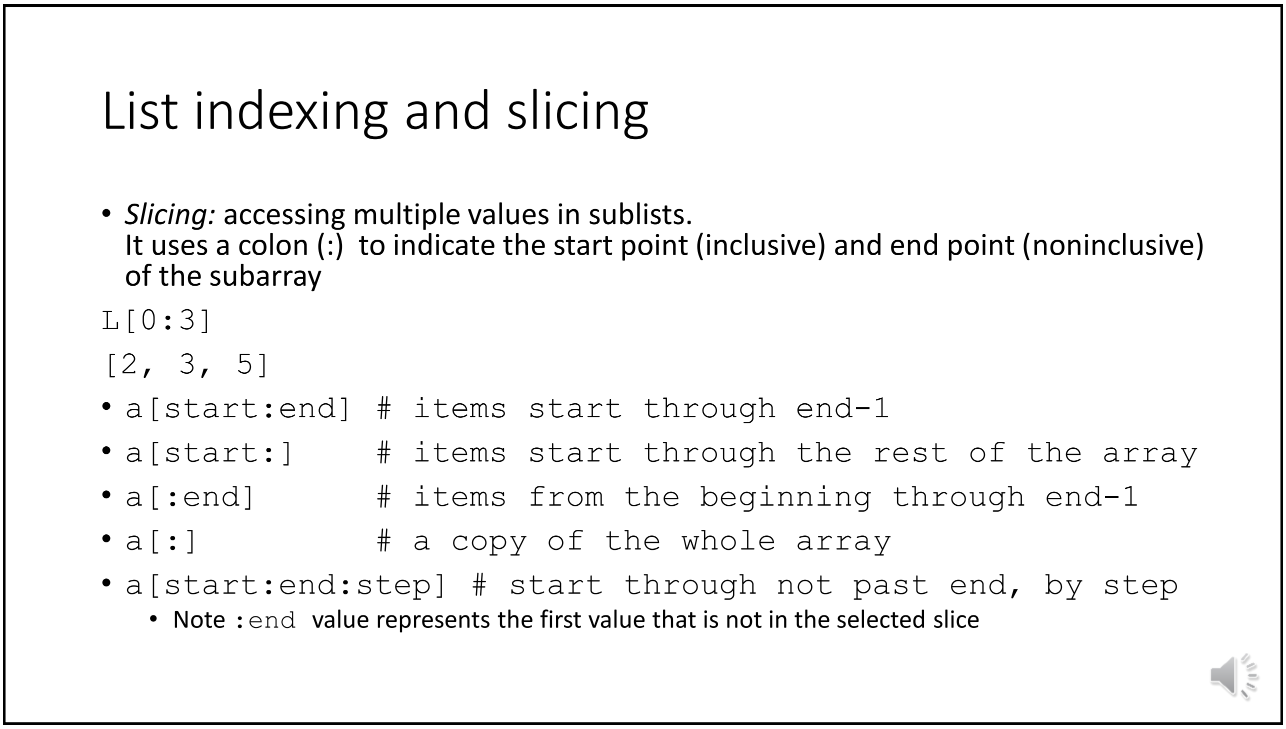
- Python uses ***zero-based*** **indexing**

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
```

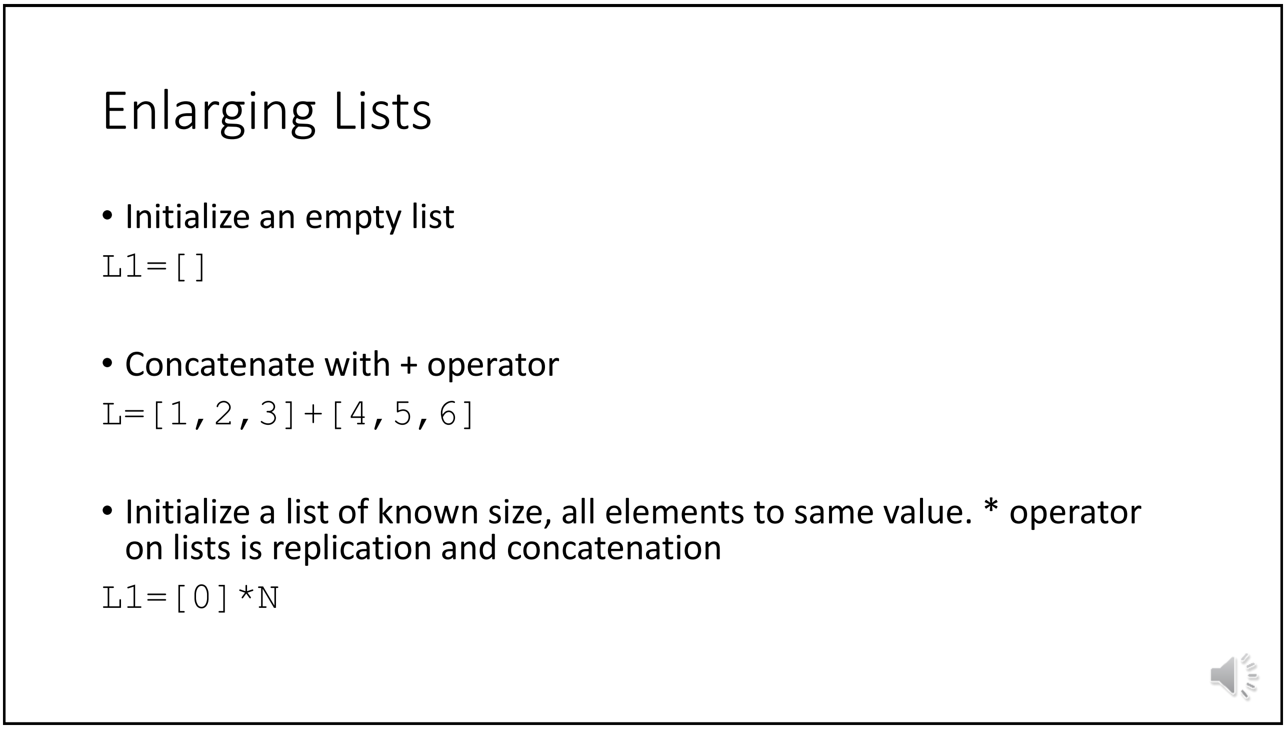


List indexing and slicing

- *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=[]
```

- Concatenate with + operator

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

- Initialize a list of known size, all elements to same value. * operator on lists is replication and concatenation

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

Membership: `in` operator

- Membership operators are used to determine if an item is in a list.
  - is a member of: `in`
  - is not a member of: `not in`

```
l3 = ['help', [1, 2], 1, 2]
1 in l3
True
'e' in l3
False
'help' in l3
True
```

Enlarging Lists

- Append

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

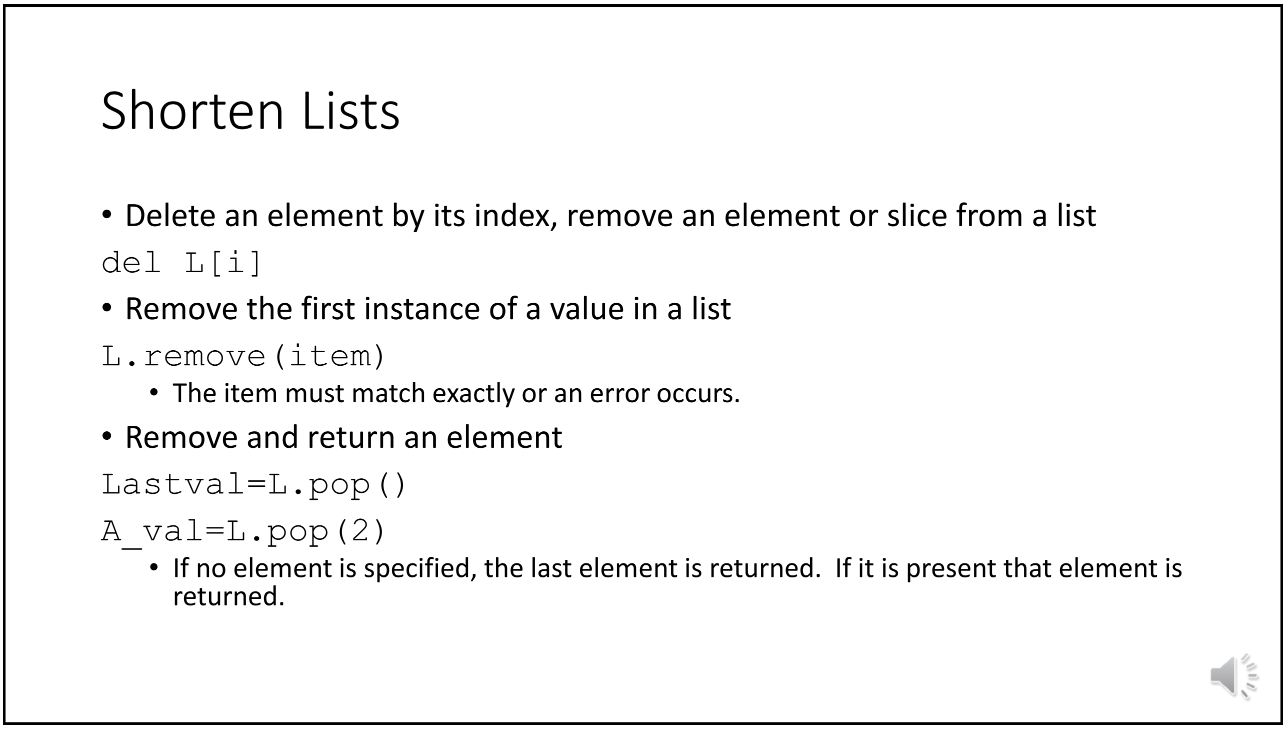
  - append adds its argument as a single element to the end of a list. The length of the list itself will increase by one.
- Extend

```
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
- 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]
```

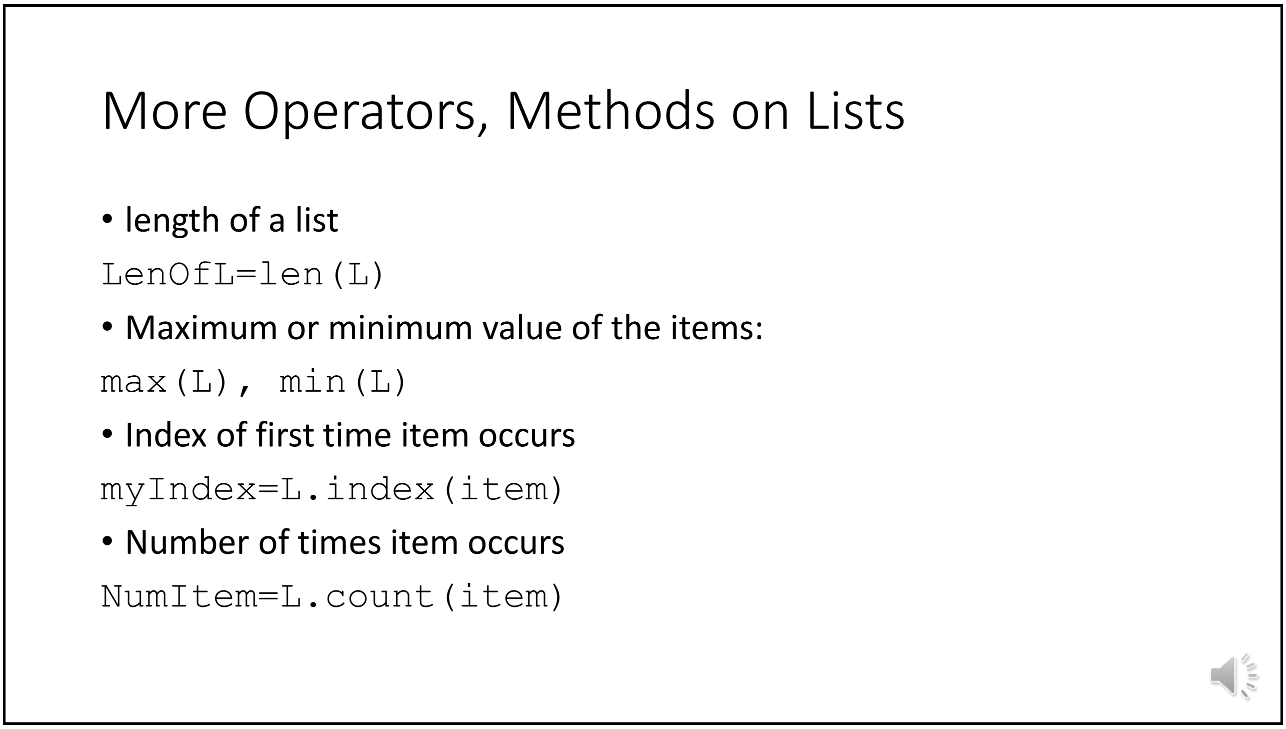
- Remove the first instance of a value in a list

```
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]
```

list()

- `list()` takes sequence types and converts them to lists.

```
In [1]: list('help')
Out[1]: ['h', 'e', 'l', 'p']
```

Range

- Ranges contain an ordered list of integers.
- 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

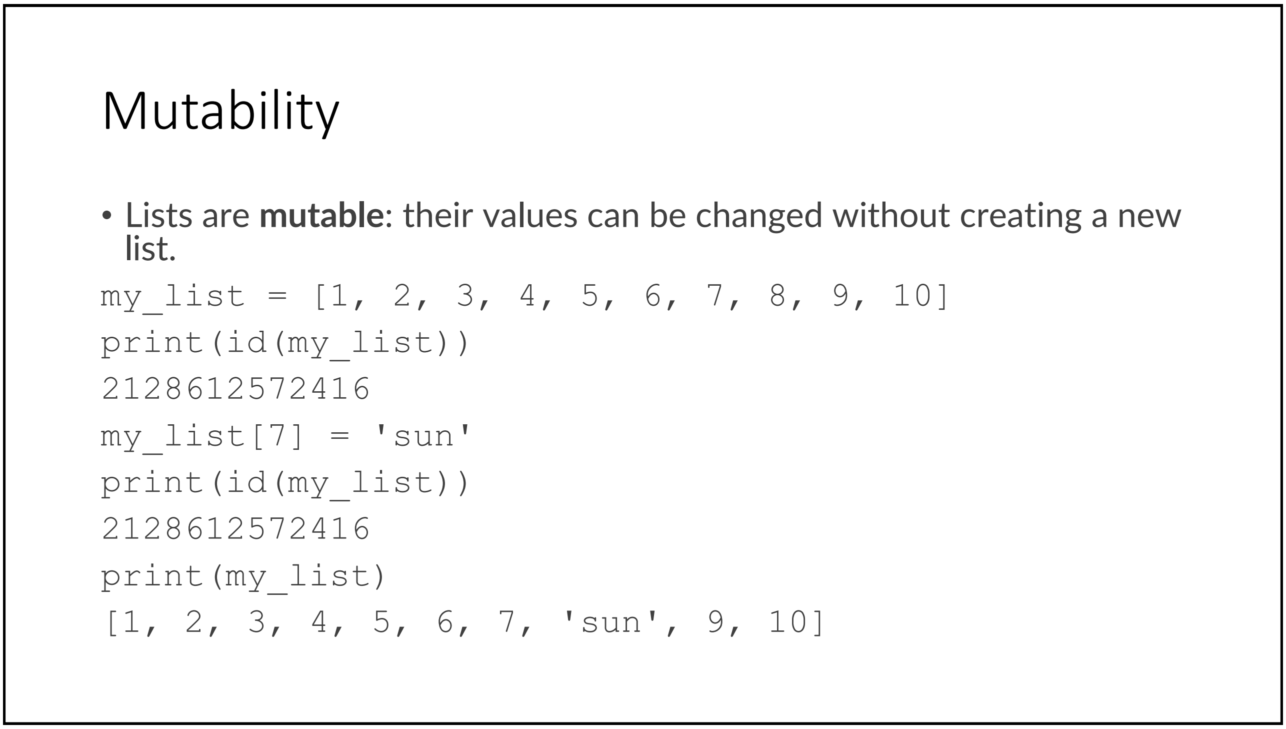
```
LR = list(range(2, 9, 3))
```

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)
```

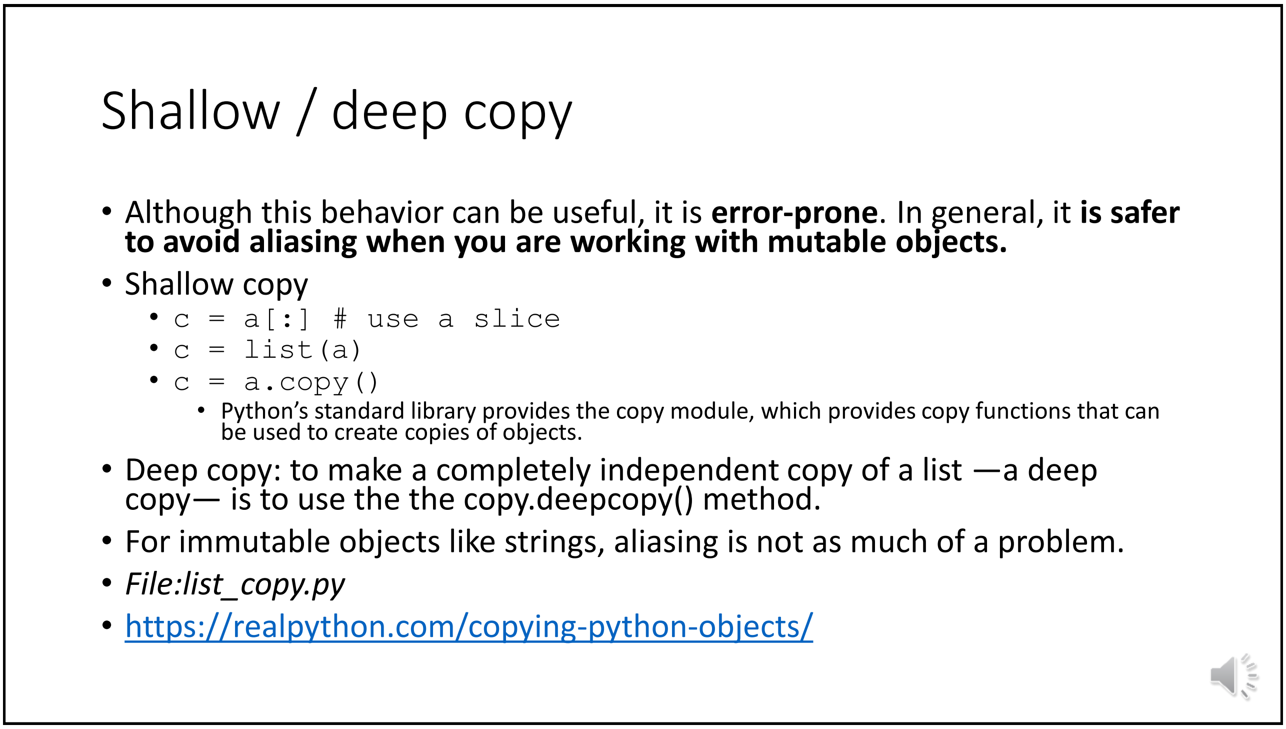
- Nested lists
  - Lists can hold any objects
  - File: list_nested.py
- Watch out for aliasing / copying lists
  - File:list_copy.py



Mutability

- Lists are **mutable**: their values can be changed without creating a new list.

```
my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(id(my_list))
2128612572416
my_list[7] = 'sun'
print(id(my_list))
2128612572416
print(my_list)
[1, 2, 3, 4, 5, 6, 7, 'sun', 9, 10]
```



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.**
- Shallow copy
  - `c = a[:] # use a slice`
  - `c = list(a)`
  - `c = a.copy()`
    - Python’s standard library provides the copy module, which provides copy functions that can be used to create copies of objects.
- Deep copy: to make a completely independent copy of a list —a deep copy— is to use the the copy.deepcopy() method.
- 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 ().

```
test_tuple_new =
('element1','element2','element3')
```

- Tuples are just like lists, except being immutable
- Because tuples are **immutable** objects, they are usually used when the list of values doesn't change.
- 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

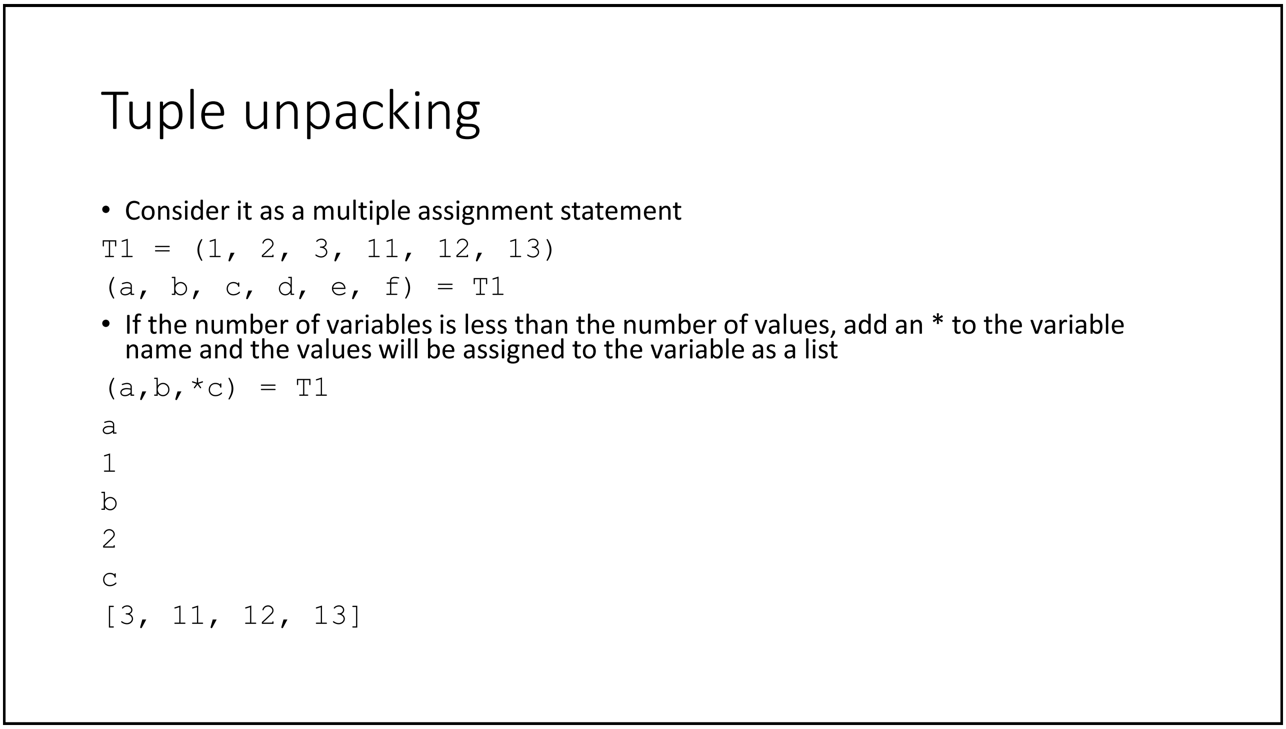
```
len(T)
```

- Concatenation (note assignment to new variable)

```
t1 = 1, 2, 3
print('id(t1)', id(t1))
2847300829504
t1 = t1 + (11, 12, 13)
print('id(t1)', id(t1))
id(t1) 2847300875648
print(t1)
(1, 2, 3, 11, 12, 13)
```

- Membership

```
3 in T
```



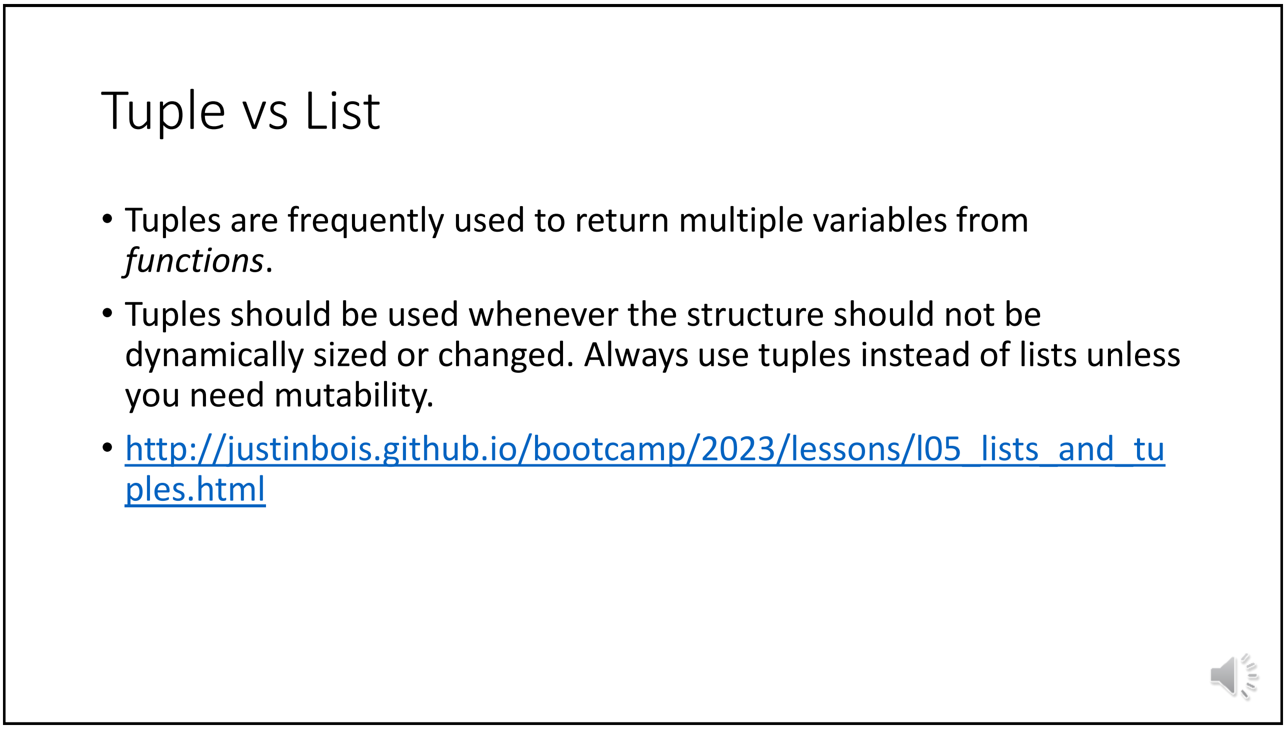
Tuple unpacking

- Consider it as a multiple assignment statement

```
T1 = (1, 2, 3, 11, 12, 13)
(a, b, c, d, e, f) = T1
```

- If the number of variables is less than the number of values, add an * to the variable name and the values will be assigned to the variable as a list

```
(a,b,*c) = T1
a
1
b
2
c
[3, 11, 12, 13]
```



Tuple vs 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. Always use tuples instead of lists unless you need mutability.
- http://justinbois.github.io/bootcamp/2023/lessons/l05_lists_and_tuples.html

Tuple vs List

- https://stackoverflow.com/questions/16940293/why-is-there-no-tuple-comprehension-in-python
- Raymond Hettinger (one of the Python core developers) had this to say about tuples in a recent tweet:
- #python tip: Generally, lists are for looping; tuples for structs. Lists are homogeneous; tuples heterogeneous. Lists for variable length.
- Although a tuple is iterable and seems like simply a immutable list, it's really the Python equivalent of a C struct:

```
struct {
    int a;
    char b;
    float c;
} foo;
struct foo x = { 3, 'g', 5.9 };
```

becomes in Python

- `x = (3, 'g', 5.9)`

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.
  - **key**: an immutable object, no need to have strings as the keys, any immutable object can be a key.
  - **value**: can be either a mutable or immutable object.
- Tip: assume that dictionaries have no sense of order (may be different depending on the Python version)

Dictionary

- Key-value pairs in a dictionary can be created using the {} notation
  - `mydict = {key : value, key : value},`
  - `mydict = {'ab' : 'abcd','cd' : 'efgh’}`
- `dict()` can also be used
  - Convert a tuple with 2-tuples into a dictionary

```
d1 = dict((('k1',11),('k2',12.3),('k3',(1, 2, 3))))
d1
{'k1': 11, 'k2': 12.3, 'k3': (1, 2, 3)}
```

- Use keyword arguments

```
d2 = dict(alfa = 1, beta = (1,2), gamma = 'help')
d2
{'alfa': 1, 'beta': (1, 2), 'gamma': 'help'}
```

Indexing a dictionary

- To access a member of the dictionary, use the key to index the content:

```
d2 = {'alfa': 1, 'beta': (1, 2), 'gamma': 'help'}
d2['beta']
(1, 2)
```

- Add new keys
  - Using Subscript notation

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

  - Using update() method

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

Dictionaries are mutable

```
print(d1)
{'k1': 11, 'k2': 12.3, 'k3': (1, 2, 3)}
d1['k1'] = 'help'
print(d1)
{'k1': 'help', 'k2': 12.3, 'k3': (1, 2, 3)}
```

Membership of dictionaries

- Operators `in` and `not in` only work on key values

```
d1
{'k1': 'help', 'k2': 12.3, 'k3': (1, 2, 3)}
'k1' in d1
True
'help' in d1
False
```

Dictionary methods

- 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: aliasing

- 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 deep copy to be sure
  - `.copy()` creates a shallow copy

Dictionary implementation

- http://justinbois.github.io/bootcamp/2023/lessons/l09_dictionaries.html
- A dictionary is an implementation of a hash table

set

set.ipynb

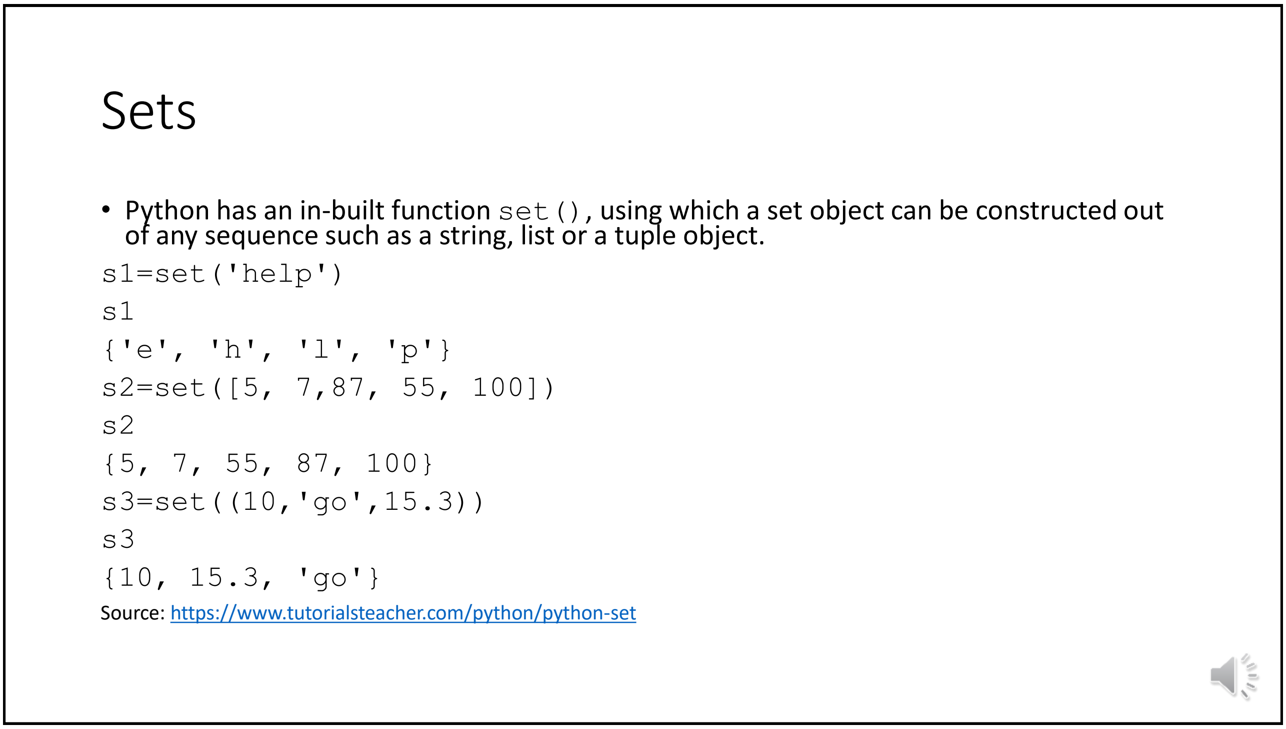
Sets

- Sets are unordered collections of distinct immutable objects.
- 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.

```
sl1 ={1, 2, [1,2]}
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-1-7a2c07e72513> in <module>
----> 1 sl1 ={1, 2, [1,2]}
TypeError: unhashable type: 'list'
```

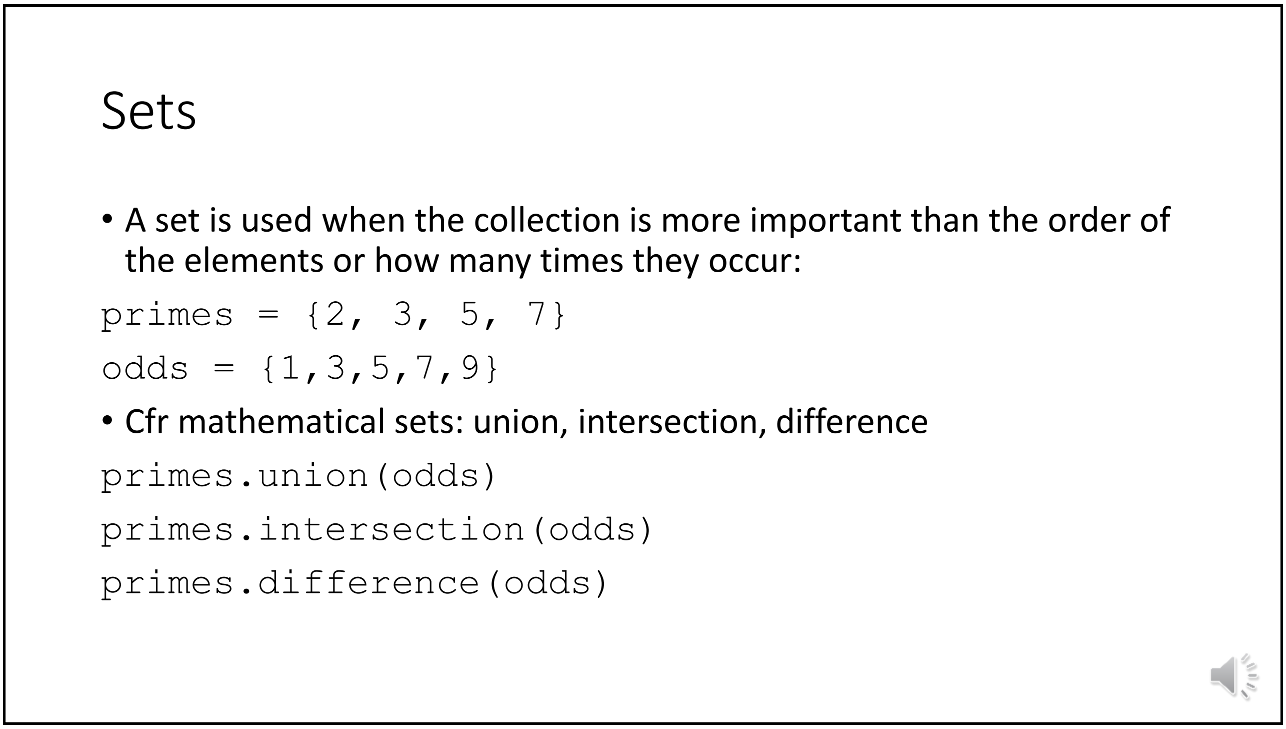


Sets

- 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.