Programming in Python

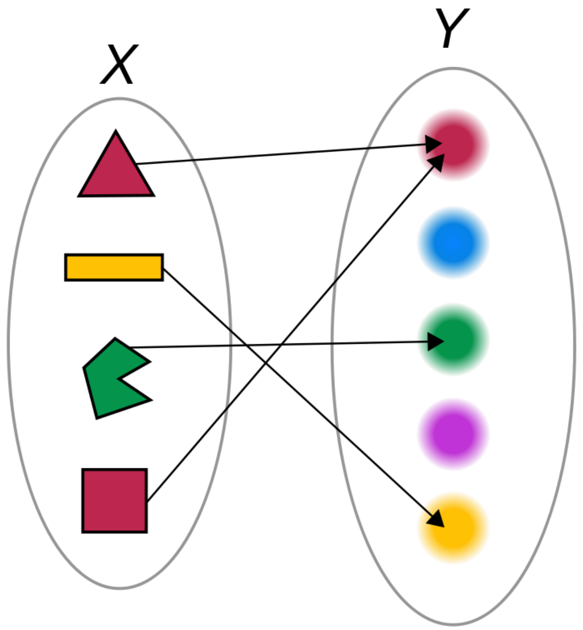
Lecture 5

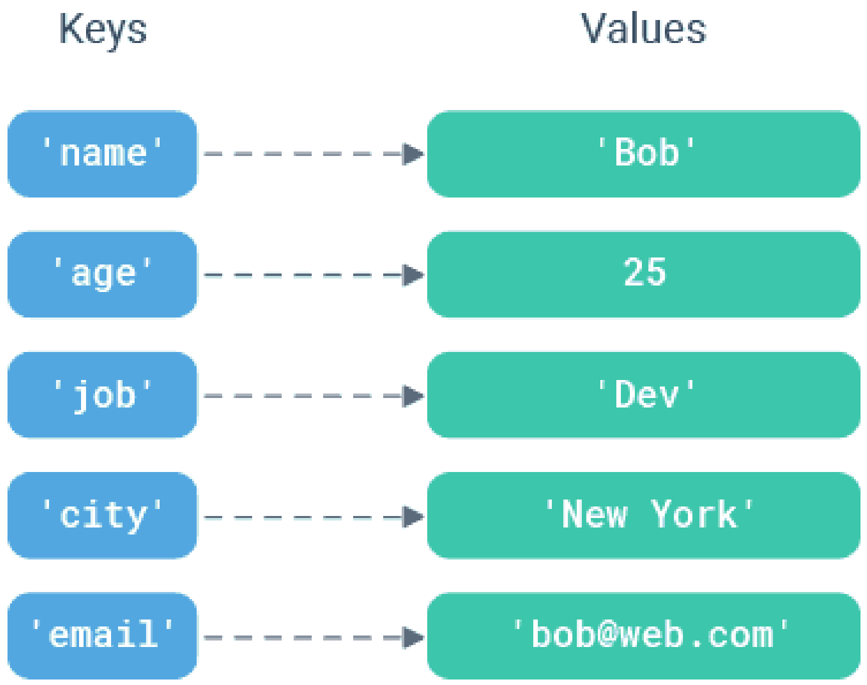
Sets. Dictionaries.

Sets & Dictionaries

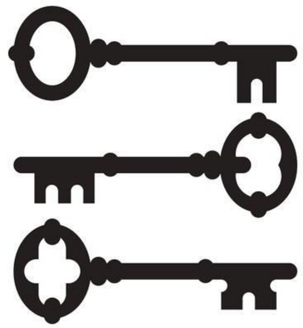
1. Dictionaries

Python dictionaries are something completely different — they are **not sequences** at all, but are known as ***mappings***.





1. Dictionaries



Mappings are also collections of other objects, but they **store objects by *key* instead of by relative position**. In fact, mappings don't maintain any reliable left-to-right order; **they simply map keys to associated values**. Dictionaries, the only mapping type in Python's core objects set, are also ***mutable***: like lists, they may be changed in place and can grow and shrink on demand.

Also like lists, they are a flexible tool for representing collections, but **their more *mnemonic* keys are better suited when a collection's items are named or labeled**—fields of a database record, for example.

Unordered collections of arbitrary objects. Keys provide the symbolic (not physical) locations of items in a dictionary.

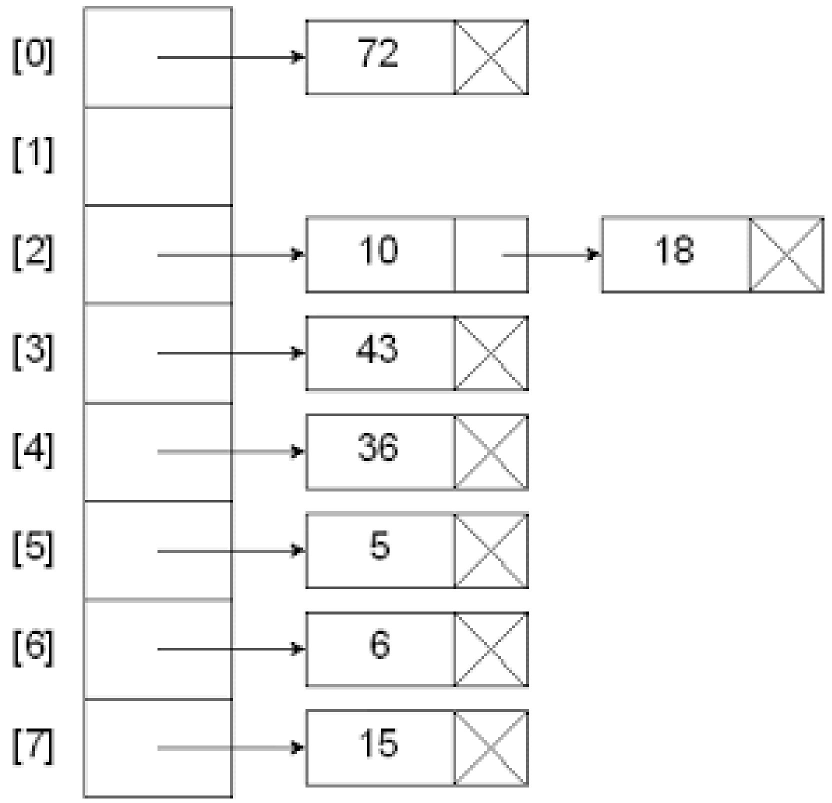
Variable-length, heterogeneous, and arbitrarily nestable. They can contain objects of any type, and they support nesting to any depth (they can contain lists, other dictionaries, and so on). ***Each key can have just one associated value***, but that value can be a *collection* of multiple objects if needed, and a given value can be stored under any number of keys.

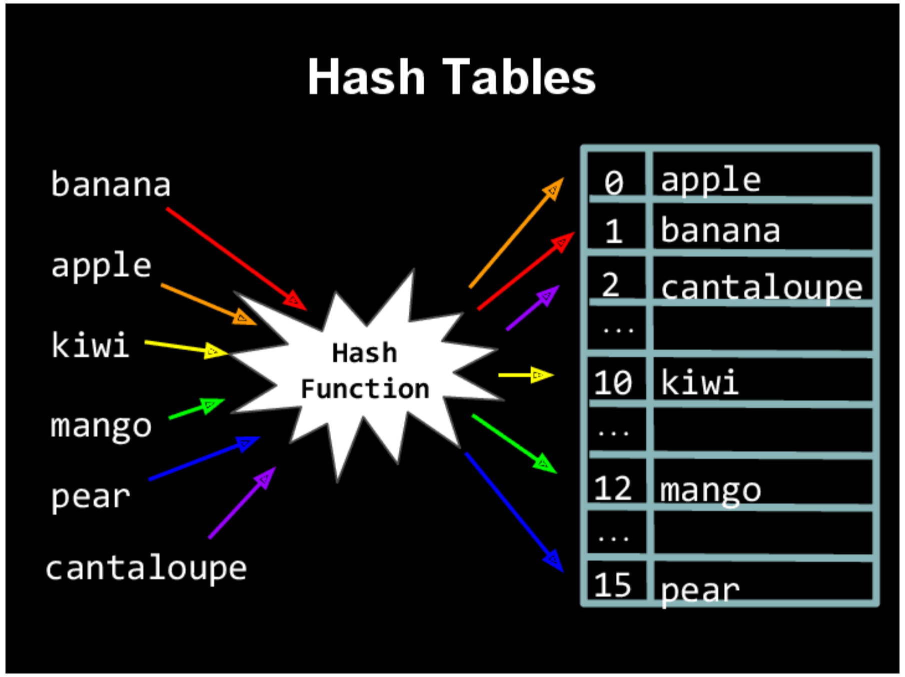
Of the category "mutable mapping". You can change dictionaries, but they don't support the sequence operations that work on strings and lists. Because **dictionaries are unordered collections**, operations that depend on a fixed positional order (e.g., concatenation, slicing) don't make sense. Dictionaries map keys to values.

Tables of object references (hash tables). If lists are arrays of object references that support access by position, dictionaries are unordered tables of object references that **support access by key**. Internally, dictionaries are implemented as ***hash tables (data structures that support very fast retrieval)***, which start small and grow on demand. Moreover, Python employs optimized hashing algorithms to find keys, so retrieval is quick. Like lists, dictionaries store object references (not copies).

Hash key = key % table size

4	=	36	%	8
2	=	18	%	8
0	=	72	%	8
3	=	43	%	8
6	=	6	%	8
2	=	10	%	8
5	=	5	%	8
7	=	15	%	8





Operation	Interpretation
D = {}	Empty dictionary
D = {'name': 'Bob', 'age': 40}	Two-item dictionary
E = {‘director': {'name': 'Bob', 'age': 40}}	Nesting
D = dict(name='Bob', age=40)	Alternative construction techniques: keywords,
D = dict([('name', 'Bob'), ('age', 40)])	key/value pairs, zipped key/value pairs, key lists
D = dict(zip(keyslist, valueslist))	
D = dict.fromkeys(['name', 'age'])	
D['name’]; E[‘director']['age']	Indexing by key
'age' in D	Membership: key present test
D.keys()	Methods: all keys
D.values()	all values
D.items()	all key+value tuples
D.copy()	copy (top-level)
D.clear()	clear (remove all items)

Operation	Interpretation
D.update(D2)	merge by keys
D.get(key, default?)	fetch by key, if absent default (or None)
D.pop(key, default?)	remove by key, if absent default (or error)
D.setdefault(key, default?)	fetch by key, if absent set default (or None)
D.popitem()	remove/return any (key, value) pair; etc.
len(D)	Length: number of stored entries
D[key] = 42	Adding/changing keys
del D[key]	Deleting entries by key
list(D.keys())	Dictionary views
D1.keys() & D2.keys()	
D = {x: x*2 for x in range(10)}	Dictionary generator

1.1. Literal

```
>>> D = {'food': 'Spam', 'quantity': 4, 'color': 'pink’}
```

Dictionaries are useful anytime we need to associate a set of values with keys—to describe the properties of something, for instance.

We can index this dictionary by key to fetch and change the keys’ associated values. The dictionary index operation uses the same syntax as that used for sequences, but the **item in the square brackets is a key, not a relative position:**

```
>>> D['food'] # Fetch value of key 'food'
'Spam'
>>> D['quantity'] += 1 # Add 1 to 'quantity' value
>>> D
{'color': 'pink', 'food': 'Spam', 'quantity': 5}
```

1.2. Other ways to built-up

Although the curly-braces literal form does see use, it is perhaps more common to see dictionaries built up in different ways (**it's rare to know all your program's data before your program runs**). The following code, for example, starts with an empty dictionary and fills it out one key at a time. ***Unlike out-of-bounds assignments in lists, which are forbidden, assignments to new dictionary keys create those keys***:

```
>>> D = {}
>>> D['name'] = 'Bob' # Create keys by assignment
>>> D['job'] = 'dev'
>>> D['age'] = 40
>>> D
{'age': 40, 'job': 'dev', 'name': 'Bob'}
>>> print(D['name'])
Bob
```

We can also make dictionaries by passing to the dict type name either *keyword arguments* (a special *name*=*value* syntax), or the result of *zipping* together sequences of keys and values obtained at runtime (e.g., from files).
Both the following make the same dictionary as the prior example and its equivalent
{} literal form, though the first tends to make for less typing:

```
>>> bob1 = dict(name='Bob', job='dev', age=40) # Keywords
>>> bob1
{'age': 40, 'name': 'Bob', 'job': 'dev’}

>>> bob2 = dict(zip(['name', 'job', 'age'], ['Bob', 'dev', 40])) # Zipping
>>> bob2
{'job': 'dev', 'name': 'Bob', 'age': 40}

>>> dict.fromkeys(['a', 'b'], 0)
{'a': 0, 'b': 0}

>>> D = dict.fromkeys('spam') # Other iterables, default value
>>> D
{'s': None, 'p': None, 'a': None, 'm': None}
```

1.3. Nesting

```
>>> rec = {'name': {'first': 'Bob', 'last': 'Smith’},
           'jobs': ['dev', 'mgr’],
           'age': 40.5}
```

We can access the components of this structure much as we did for our listbased matrix earlier, but this time most indexes are dictionary keys, not list offsets:

```
>>> rec['name'] # 'name' is a nested dictionary
{'last': 'Smith', 'first': 'Bob'}
```

```
>>> rec['name']['last'] # Index the nested dictionary
'Smith'
```

```
>>> rec['jobs'] # 'jobs' is a nested list
['dev', 'mgr']
>>> rec['jobs'][-1] # Index the nested list
'mgr'
```

```
>>> rec['jobs'].append('janitor') # Expand Bob's job description in place
>>> rec
{'age': 40.5, 'jobs': ['dev', 'mgr', 'janitor'], 'name': {'last': 'Smith',
'first': 'Bob'}}
```

1.4. Generators

```
>>> D = {x: x ** 2 for x in [1, 2, 3, 4]} # Or: range(1, 5)
>>> D
{1: 1, 2: 4, 3: 9, 4: 16}

>>> D = {c: c * 4 for c in 'SPAM'} # Loop over any iterable
>>> D
{'S': 'SSSS', 'P': 'PPPP', 'A': 'AAAA', 'M': 'MMMM’}

>>> D = {k: v for (k, v) in zip(['a', 'b', 'c'], [1, 2, 3])}
>>> D
{'b': 2, 'c': 3, 'a': 1}

>>> D = {k: None for k in 'spam'}
>>> D
{'s': None, 'p': None, 'a': None, 'm': None}
```

1.5. Using dictionaries to simulate flexible lists: Integer keys

When you use lists, it is illegal to assign to an offset that is off the end of the list:

```
>>> L = []
>>> L[99] = 'spam'
Traceback (most recent call last):
File "<stdin>", line 1, in ?
IndexError: list assignment index out of range
```

Although you can use repetition to preallocate as big a list as you'll need (e.g., [0]*100), you can also do something that looks similar with dictionaries that does not require such space allocations. By using integer keys, dictionaries can emulate lists that seem to grow on offset assignment:

```
>>> D = {}
>>> D[99] = 'spam'
>>> D[99]
'spam'
>>> D
{99: 'spam'}
```

1.6. Using dictionaries for sparse data structures: Tuple keys

In a similar way, dictionary keys are also commonly leveraged to implement *sparse* data structures—for example, multidimensional arrays where only a few positions have values stored in them:

```
>>> Matrix = {}
>>> Matrix[(2, 3, 4)] = 88
>>> Matrix[(7, 8, 9)] = 99
>>>
>>> X = 2; Y = 3; Z = 4 # ; separates statements
>>> Matrix[(X, Y, Z)]
>>> Matrix
{(2, 3, 4): 88, (7, 8, 9): 99}

>>> Matrix[(2,3,6)]
Traceback (most recent call last):
File "<stdin>", line 1, in ?
KeyError: (2, 3, 6)
```

1.7. Avoiding missing-key errors

There are at least three ways to fill in a default value instead of getting such an error message:

```
>>> if (2, 3, 6) in Matrix: # Check for key before fetch
        print(Matrix[(2, 3, 6)])
    else:
        print(0)
0

>>> try:
         print(Matrix[(2, 3, 6)]) # Try to index
    except KeyError: # Catch and recover
         print(0)

>>> Matrix.get((2, 3, 4), 0) # Exists: fetch and return
88
>>> Matrix.get((2, 3, 6), 0) # Doesn't exist: use default arg
0
```

2. Sets

The ***set***—**an unordered collection of unique and immutable objects** that supports operations corresponding to mathematical set theory.

- By definition, **an item appears only once in a set**, no matter how many times it is added.
- Sets are iterable, can grow and shrink on demand, and may contain a variety of object types.
- **Sets are neither sequence nor mapping types.**
- Sets are mutable, set's items are immutable.

```
set([1, 2, 3, 4]) # Built-in call
{1, 2, 3, 4} # Newer set literals
```

2.1. Common mathematical set operations

```python
x = set('abcde')
y = set('bdxyz')

print('x - y = ', x - y)  # Difference
print('x | y = ', x | y)  # Union
print('x & y = ', x & y)  # Intersection
print('x ^ y = ', x ^ y)  # Symmetric difference (XOR)
print('x > y = ', x > y)  # Superset
print('x < y = ', x < y)  # Subset
print('"a" in x ', "a" in x)  # Membership
```

```
x - y =  {'e', 'c', 'a'}
x | y =  {'c', 'd', 'x', 'z', 'y', 'b', 'a', 'e'}
x & y =  {'b', 'd'}
x ^ y =  {'c', 'x', 'z', 'y', 'a', 'e'}
x > y =  False
x < y =  False
"a" in x  True
```

2.2. Usage

Because items are stored only once in a set, sets can be used to ***filter duplicates*** out of other collections, though items may be reordered in the process because sets are unordered in general.

```
>>> L = [1, 2, 1, 3, 2, 4, 5]
>>> set(L)
{1, 2, 3, 4, 5}
>>> L = list(set(L)) # Remove duplicates
>>> L
[1, 2, 3, 4, 5]
```

```
>>> list(set(['yy', 'cc', 'aa', 'xx', 'dd', 'aa'])) # But order may change
['cc', 'xx', 'yy', 'dd', 'aa']
```

Sets can be used to *isolate differences* in lists, strings, and other iterable objects too—simply convert to sets and take the difference—though again the unordered nature of sets means that the results may not match that of the originals.

```
>>> set([1, 3, 5, 7]) - set([1, 2, 4, 5, 6]) # Find list differences
{3, 7}

>>> set('abcdefg') - set('abdghij') # Find string differences
{'c', 'e', 'f’}

>>> set('spam') - set(['h', 'a', 'm']) # Find differences, mixed
{'p', 's'}
```

You can also use sets to perform *order-neutral equality* tests by converting to a set before the test, because order doesn’t matter in a set. More formally, two sets are *equal* if and only if every element of each set is contained in the other—that is, each is a subset of the other, regardless of order.
For instance, you might use this to compare the outputs of programs that should work the same but may generate results in different order.

```
>>> L1, L2 = [1, 3, 5, 2, 4], [2, 5, 3, 4, 1]
>>> L1 == L2 # Order matters in sequences
False

>>> set(L1) == set(L2) # Order-neutral equality
True

>>> sorted(L1) == sorted(L2) # Similar but results ordered
True

>>> 'spam' == 'asmp', set('spam') == set('asmp'), sorted('spam') == sorted('asmp')
(False, True, True)
```

2.3. Frozenset

Frozenset is an immutable variant of set.
In **sets**, you can change the items when you required, whereas items of the **frozenset** remain the same once created. Because of this, frozen sets are used as a key in Dictionary:

```
Student = {"name": "Ankit", "age": 21, "sex": "Male",
           "college": "MNNIT Allahabad", "address": "Allahabad"}
key = frozenset(Student)
print('The frozen set is:', key)
The frozen set is: frozenset({'name', 'college', 'address', 'sex', 'age'})
```

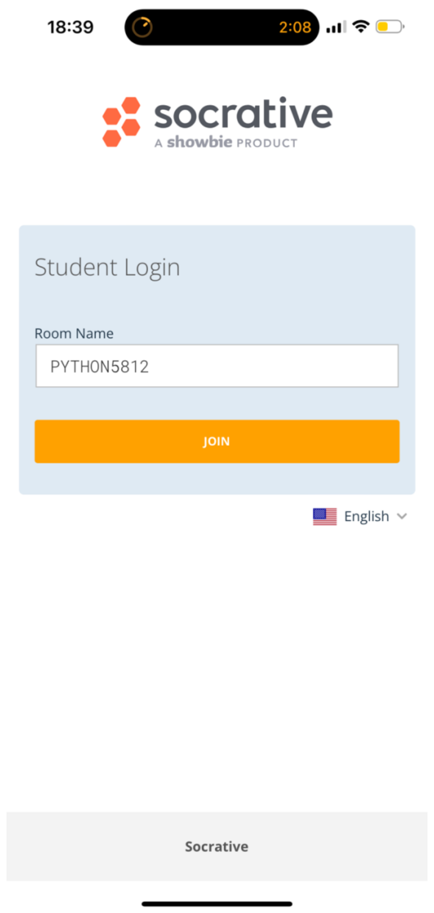
Quiz with Socrative

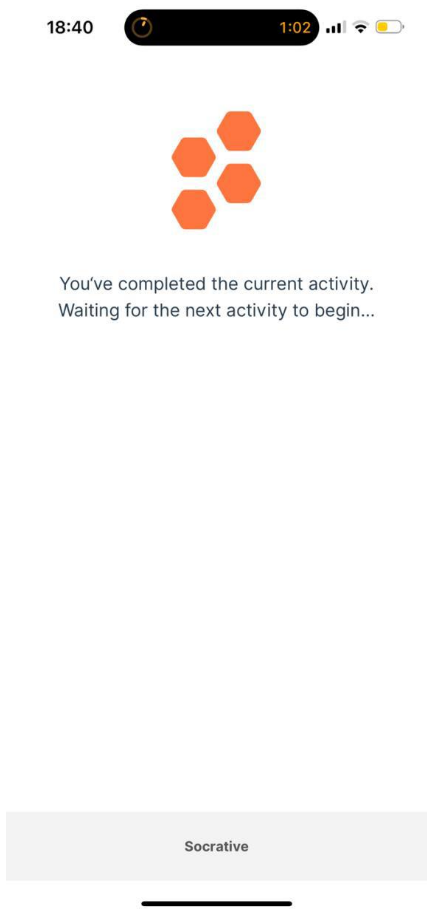
1. Install mobile application **Socrative Student** or go to the https://www.socrative.com/

2. Join Room name **PYTHON5812**

3. Put your Name in the format:
   Фамилия_Имя_подгруппа

   Surname_name_subgroup number





You've completed the current activity.
Waiting for the next activity to begin...