

# Scientific Programming with Python

## Dictionaries and Sets

UNIVERSITY of  
**HOUSTON**

DIVISION OF RESEARCH  
HEWLETT PACKARD ENTERPRISE DATA SCIENCE INSTITUTE

# Dictionaries

- *“Dictionary: a book, optical disc, mobile device, or online lexical resource (such as Dictionary.com) containing a selection of the words of a language, giving information about their meanings, pronunciations, etymologies, inflected forms, derived forms, etc., expressed in either the same or another language; lexicon; glossary”*
- *“a dictionary is a collection of words matched with their definitions”*
- Python has a built in dictionary type/data structure called **dict** which you can use to create dictionaries with arbitrary definitions

# So Far

- Lists, Strings are sequential
- Dictionaries are **mapped types**
- The mapping is done from a **key**, immutable type, to a **value**, any Python data type

# Dictionary Basics

- How to create an empty dictionary:  
`acronyms = {}`
- To add key-value pairs:
  - `acronyms['lol'] = "laughing out loud"`  
`acronyms['tty'] = "talk to you later"`  
`acronyms['bfm'] = "bye for now"`
- Alternative way to create a dictionary:
  - `acronyms = {'bfm': 'bye for now', 'tty': 'talk to you later', 'lol': 'laughing out loud'}`

Order is not important!

# Dictionary: Basics

A collection of (key:value) pairs

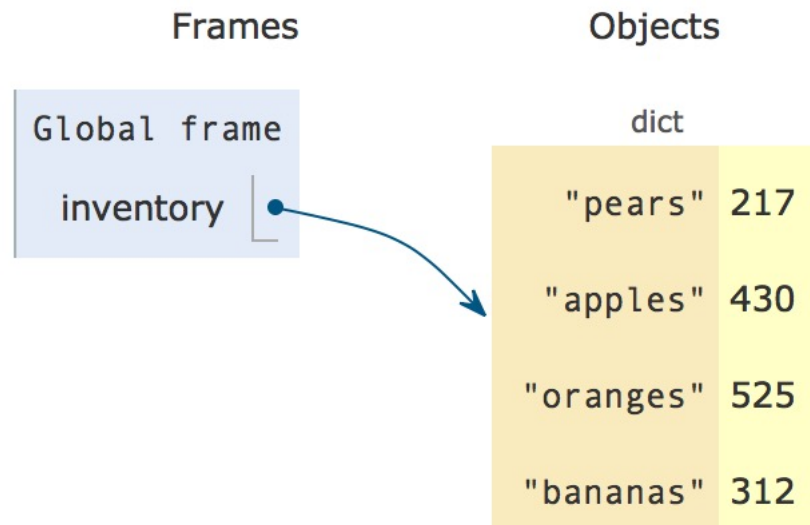
Example: Store the employee records indexed by SS#

```
>> Employees = {456965079: 'David Smith',  
                823878356: 'Lisa Miller',  
                813756744: 'Pat Murphy' }  
>>Employees[813756744]
```

'Pat Murphy'

# Another Example

```
inventory = {'apples': 430, 'bananas': 312, 'oranges': 525, 'pears': 217}  
inventory['bananas'] = inventory['bananas'] + 200
```





# Properties of Dictionaries

- Not ordered
- Mutable
  - (key,value) pairs can be added  
`dict[k] = newvalue`
  - Value can be modified  
`dict[k] = newvalue`
  - Values can be deleted  
`del dict[k]`  
`dict.pop(k)` also returns the value
  - The key itself is immutable

# Dictionary Methods

Method	Parameters	Description
keys	none	Returns a view of the keys in the dictionary
values	none	Returns a view of the values in the dictionary
items	none	Returns a view of the key-value pairs in the dictionary
get	key	Returns the value associated with key; None otherwise
get	key,alt	Returns the value associated with key; alt otherwise



# Using Dictionary Methods

- `inventory = {'apples': 430, 'bananas': 312, 'oranges': 525, 'pears': 217}`
- `for akey in inventory.keys():`
- `print("Got key", akey, "which maps to value", inventory[akey])`

```
ks = list(inventory.keys())  
print(ks)
```

The method `keys()` will return an object **view**

# Iterating over Dictionaries

- We can simply use the for loop:

- Employees = {  
    '456965079': 'David Smith',  
    '823878356': 'Lisa Miller',  
    '813756744': 'Pat Murphy'}

```
for k in Employees:  
    print("Got key", k)
```

# Other uses

- We can use the operators `in` and `not in` to test if a key value exists in the dictionary:
- `if '456965079' in Employees:`  
    `print(Employees['456965079'])`  
    `else:`  
        `print("This SSN doesn't exist in our data base")`

# Aliasing and Copying

- Dictionaries are mutable objects so you should expect the same behavior we learned with lists.
- `opposites = {'up': 'down', 'right': 'wrong', 'true': 'false'}`  
`alias = opposites`

```
print(alias is opposites)
```

```
alias['right'] = 'left'  
print(opposites['right'])
```

# Aliasing and Copying

- If you really want to create a copy of the dictionary use the dictionary copy method:

- `opposites = {'up': 'down', 'right': 'wrong', 'true': 'false'}`  
`acopy = opposites.copy()`

```
print(acopy is opposites)
```

```
acopy['right'] = 'left'  
print(opposites['right'])
```

# Sparse Matrices

- Matrix: two dimensional collection, we can think of it as a list of lists.

```
matrix = [[0, 0, 0, 1, 0],  
          [0, 0, 0, 0, 0],  
          [0, 2, 0, 0, 0],  
          [0, 0, 0, 0, 0],  
          [0, 0, 0, 3, 0]]
```

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 \end{bmatrix}$$

Most of these entries are zeros!



# Using a Dictionary to Represent Sparse Matrices

- `matrix = {(0, 3): 1, (2, 1): 2, (4, 3): 3}`
- To access an element in this matrix we use the `[]` operator:
- `matrix[(0, 3)]`

We use only one index, a tuple, to access elements

- Note that trying to access a zero element will cause errors. Instead we use the `get` method:
- `matrix = {(0, 3): 1, (2, 1): 2, (4, 3): 3}`  
`print(matrix.get((1, 3), 0))`

# Sets

- The [sets](#) module provides classes for constructing and manipulating unordered collections of unique elements. Common uses include membership testing, removing duplicates from a sequence, and computing standard math operations on sets such as intersection, union, difference, and symmetric difference.
- Like other collections, sets support `x in set`, `len(set)`, and `for x in set`. Being an unordered collection, sets do not record element position or order of insertion. Accordingly, sets do not support indexing, slicing, or other sequence-like behavior.
- Set syntax
  - `a = {1, 4, 5}`
  - Empty set
  - `a = set()`

# Set Operations

Operation	Equivalent	Result
<code>len(s)</code>		number of elements in set <i>s</i> (cardinality)
<code>x in s</code>		test <i>x</i> for membership in <i>s</i>
<code>x not in s</code>		test <i>x</i> for non-membership in <i>s</i>
<code>s.issubset(t)</code>	<code>s &lt;= t</code>	test whether every element in <i>s</i> is in <i>t</i>
<code>s.issuperset(t)</code>	<code>s &gt;= t</code>	test whether every element in <i>t</i> is in <i>s</i>
<code>s.union(t)</code>	<code>s   t</code>	new set with elements from both <i>s</i> and <i>t</i>
<code>s.intersection(t)</code>	<code>s &amp; t</code>	new set with elements common to <i>s</i> and <i>t</i>
<code>s.difference(t)</code>	<code>s - t</code>	new set with elements in <i>s</i> but not in <i>t</i>
<code>s.symmetric_difference(t)</code>	<code>s ^ t</code>	new set with elements in either <i>s</i> or <i>t</i> but not both
<code>s.copy()</code>		new set with a shallow copy of <i>s</i>

# Examples

```
engineers = {'John', 'Jane', 'Jack', "Janice"}
```

```
programmers = {'Jack', 'Sam', 'Susan', 'Janice'}
```

```
managers = {'Jane', 'Jack', 'Susan', 'Zack'}
```

```
#Compute union
```

```
employees = engineers | programmers | managers
```

```
#Compute intersection
```

```
engineering_management = engineers & managers
```

```
fulltime_management = managers - engineers - programmers
```

```
engineers.add('Marvin') # add element
```

```
print (engineers)
```

```
employees.issuperset(engineers) # superset test
```

```
False
```

```
employees.update(engineers) # update from another set
```

```
employees.issuperset(engineers)
```

```
True
```