Computational Thinking with Programming

@cse_bennett @csebennett

Lecture Contents

- **Non Sequential Collections**
- Dictionary
- Dictionary Operations

Non-Sequential Collections

- Likes List and tuples are the sequential collections where elements are ordered accessed through by their indexes.

- Python has two types of non-sequential collections.
  - Dictionaries
  - Sets

Python Dictionaries

- Dictionaries in Python provides a concept of *associative data structure,* where the elements of are unordered and accessed by an associated key value instead of index.
- A dictionary is a collection which is unordered, changeable (mutable) and indexed. In Python dictionaries are written with curly brackets, and they have keys and values.
- **Syntax for declaring dictionaries in Python:**

```
                       key    Value
                        ↓       ↓
daily_temps = {
                    'sun’:   68.8,
                    'mon’:  70.2, 'tue': 67.2,
                    'wed’:  71.8, 'thur': 73.2,
                    'fri’:      75.6, 'sat': 74.0
               }
```

```
#Example :Create and print a dictionary
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(thisdict)
```

```
Output:
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
```

Accessing Items in Dictionaries

- You can access the items of a dictionary by referring to its key name, inside square brackets **or** using get() method by passing key name:

```
#Get the value of the "model" key
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
x = thisdict["model"]
print(x)
```

```
#Geting value of "model" key using get():

x = thisdict.get("model")
```

```
Output is same in both case:

Mustang
```

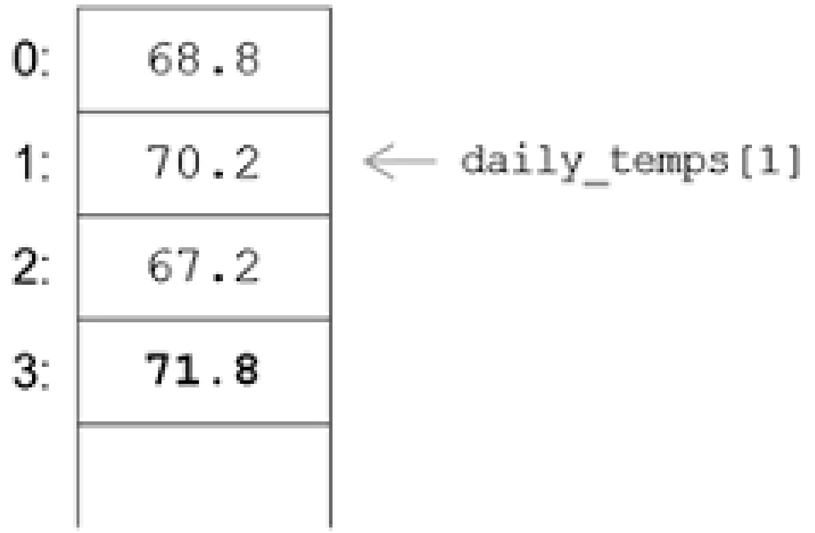
- **Change Values:** You can change the value of a specific item by referring to its key name:

```
#Example: Change the "year" to 2018:

thisdict["year"] = 2018
print(thisdict)
```

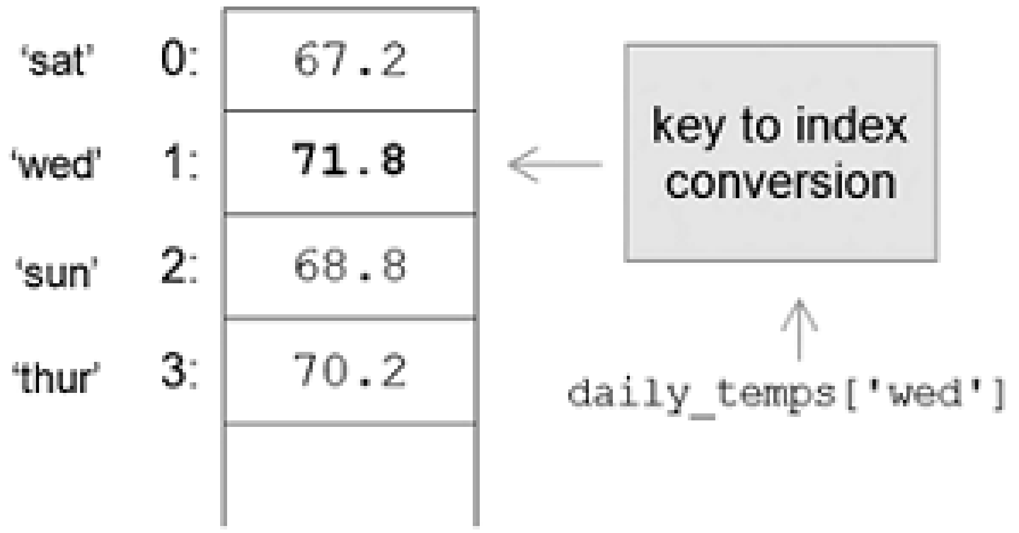
```
Output:

{'brand': 'Ford', 'model': 'Mustang', 'year': 2018}
```

Indexed vs. Associative Data Structure

- The syntax for accessing an element of a dictionary is the same as for accessing elements of sequence types, except that a key value is used within the square brackets instead of an index value: *daily_temps['sun']*

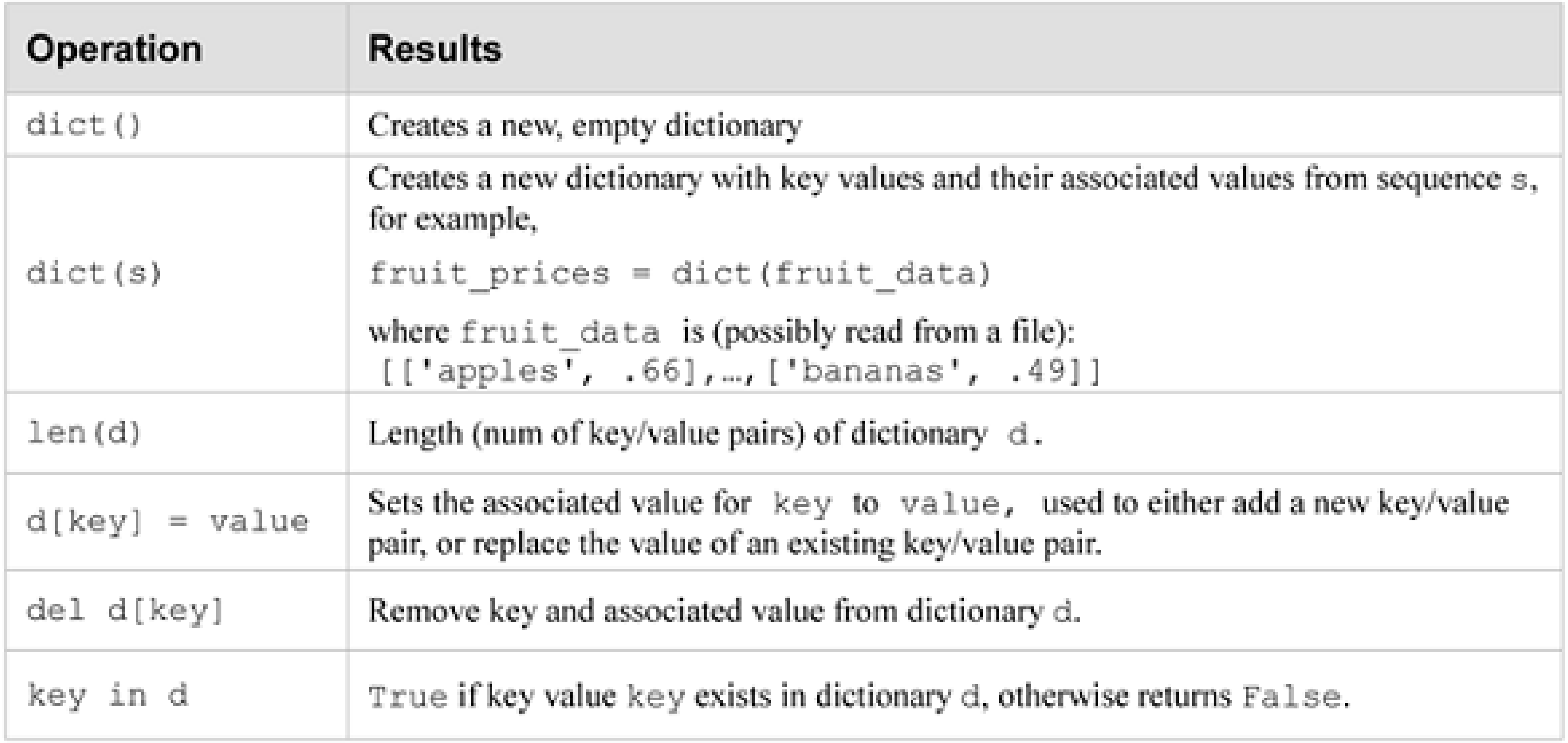


indexed data structure



associative data structure

Operations for Dynamically Manipulating Dictionaries



Operation	Results
`dict()`	Creates a new, empty dictionary

`dict(s)`	Creates a new dictionary with key values and their associated values from sequence `s`, for example,
	`fruit_prices = dict(fruit_data)`
	where `fruit_data` is (possibly read from a file):
	`[['apples', .66],…,['bananas', .49]]`

`len(d)`	Length (num of key/value pairs) of dictionary `d`.
`d[key] = value`	Sets the associated value for `key` to `value`, used to either add a new key/value pair, or replace the value of an existing key/value pair.
`del d[key]`	Remove key and associated value from dictionary `d`.
`key in d`	`True` if key value `key` exists in dictionary `d`, otherwise returns `False`.

Loop Through a Dictionary

- You can loop through a dictionary by using a for loop.
- When looping through a dictionary, the return value are the *keys* of the dictionary.

```
#Print all key names in the dictionary, one by one:
for x in thisdict:
  print(x)
```

```
Output:
brand
model
year
```

- Python also provides methods to return the *values* as well.

```
#Print all values in the dictionary, one by one:
for x in thisdict:
  print(thisdict[x])

#You can also use the values() method to return values of
a dictionary:
for x in thisdict.values():
  print(x)
```

```
Output:
Ford
Mustang
1964
```

Loop Through a Dictionary

Loop through both *keys* and *values*, by using the `items()` method:

```
#Example:

thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}

for x, y in thisdict.items():
  print(x, y)
```

```
Output:

brand Ford
model Mustang
year 1964
```

Dictionary Checking and Length

- To determine if a specified key is present in a dictionary use the in keyword:

#Example: Check if "model" is present in the dictionary

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
if "model" in thisdict:
  print("Yes, 'model' is one of the keys in the thisdict
dictionary")
```

```
Output:

Yes, 'model' is one of the keys in the thisdict
dictionary
```

- To determine how many items (key-value pairs) a dictionary has, use the *len()* function.

#Print the number of items in the dictionary:

```
print(len(thisdict))
```

```
Output:
3
```

Adding and Removing Items in Dictionary

- **Adding Items:** It is done by using a new index key and assigning a value to it.

```
#Example:
thisdict = {"brand": "Ford",  "model": "Mustang",  "year": 1964}
thisdict["color"] = "red"
print(thisdict)
```

```
Output: {'brand': 'Ford’, 'model': 'Mustang’, 'year': 1964, 'color': 'red'}
```

- **Removing Items:** There are several methods to remove items from a dictionary.
  - **pop():** Removes the item with the specified key name
  - **popitem():** Removes the last inserted item (in versions before 3.7, a random item is removed instead).
  - **del keyword:** Removes the item with the specified key name as well removes the dictionary completely.
  - **clear():** It empties the dictionary.

Removing Items from Dictionary

Example1: pop()

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.pop("model")
print(thisdict)
```

Example2: popitem()

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.popitem()
print(thisdict)
```

Example3: del keyword

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
del thisdict["model"]
print(thisdict)
```

Output: Example1

```
{'brand': 'Ford', 'year': 1964}
```

Output: Example2

```
{'brand': 'Ford', 'model': 'Mustang'}
```

Output: Example3

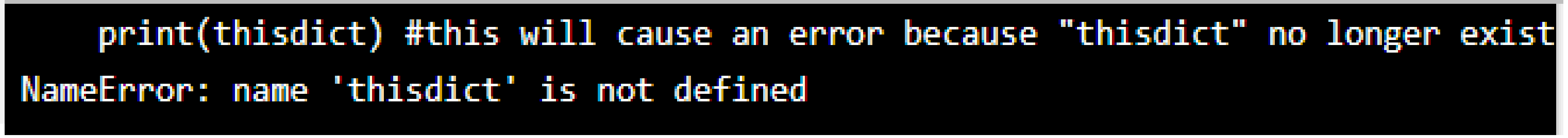
```
{'brand': 'Ford', 'year': 1964}
```

Delete or Empties Dictionary

Example1: del for deleting dictionary

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
del thisdict
print(thisdict)
```

Output:



```
    print(thisdict) #this will cause an error because "thisdict" no longer exist
NameError: name 'thisdict' is not defined
```

Example2: clear()

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.clear()
print(thisdict)
```

Output:

```
{}
```

Copy Dictionaries

- Dictionary cannot be copied simply by typing dict2 = dict1, **because:** dict2 will only be a *reference* to dict1, and changes made in dict1 will automatically also be made in dict2.

- There are ways to make a copy, one way is to use the built-in Dictionary method **copy()**.

```
thisdict = { "brand": "Ford", "model": "Mustang",  "year": 1964 }
mydict = thisdict.copy()
print(mydict)
```

```
Output: {'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
```

- Another way to make a copy is to use the built-in function **dict().**

```
thisdict = {  "brand": "Ford",   "model": "Mustang",   "year": 1964 }
mydict = dict(thisdict)
print(mydict)
```

```
Output: {'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
```

Nested Dictionaries

- A dictionary can also contain many dictionaries, this is called nested dictionaries.

#Example: Create a dictionary that contain three dictionaries

```
myfamily = {
  "child1" : {
    "name" : "Emil",
    "year" : 2004
  },
  "child2" : {
    "name" : "Tobias",
    "year" : 2007
  },
  "child3" : {
    "name" : "Linus",
    "year" : 2011
  }
}
print(myfamily)
```

Output:

```
{
'child1': {'name': 'Emil', 'year': 2004},
'child2': {'name': 'Tobias', 'year': 2007},
'child3': {'name': 'Linus', 'year': 2011}
}
```

Nested Dictionaries

- You can also do the nesting of three dictionaries that already exists as dictionaries.

```
#Create three dictionaries, then create one dictionary that will contain the other three dictionaries:
child1 = {
  "name" : "Emil",
  "year" : 2004
}
child2 = {
  "name" : "Tobias",
  "year" : 2007
}
child3 = {
  "name" : "Linus",
  "year" : 2011
}
myfamily = {
  "child1" : child1,
  "child2" : child2,
  "child3" : child3
}
print(myfamily)
```

Output:

```
{
'child1': {'name': 'Emil', 'year': 2004},
'child2': {'name': 'Tobias', 'year': 2007},
'child3': {'name': 'Linus', 'year': 2011}
}
```

The dict() Constructor

- It is also possible to use the dict() constructor to make a new dictionary.

Example:

```
thisdict = dict(brand="Ford", model="Mustang", year=1964)
print(thisdict)
```

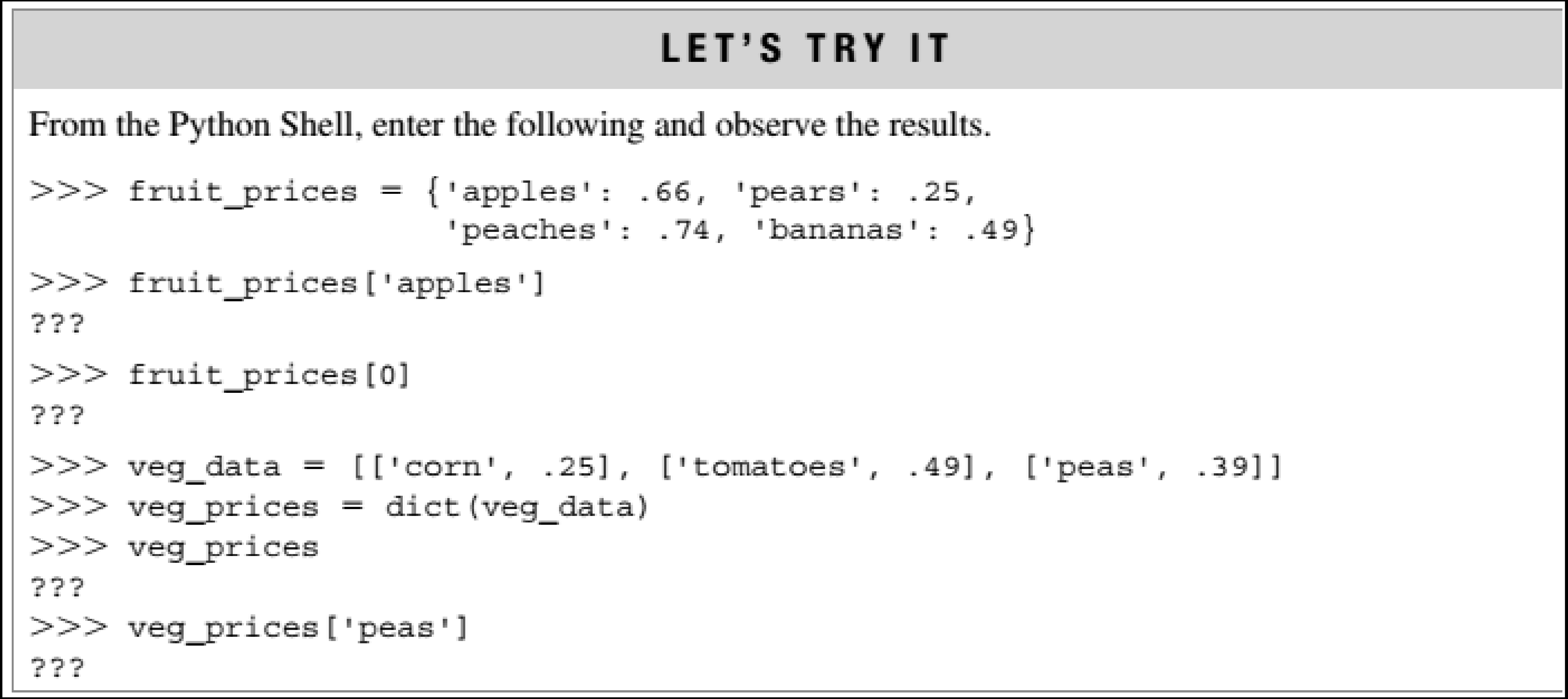
```
Output: {'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
```

Note: ***1. Note that here keywords are not string literals.***
2. Note that the use of equals rather than colon for the assignment.

Dictionary Methods

(Python provides a several of built-in methods that you can use on dictionaries.)

Method	Description
clear()	Removes all the elements from the dictionary
copy()	Returns a copy of the dictionary
fromkeys()	Returns a dictionary with the specified keys and value
get()	Returns the value of the specified key
items()	Returns a list containing a tuple for each key value pair
keys()	Returns a list containing the dictionary's keys
pop()	Removes the element with the specified key
popitem()	Removes the last inserted key-value pair
setdefault()	Returns the value of the specified key. If the key does not exist: insert the key, with the specified value
update()	Updates the dictionary with the specified key-value pairs
values()	Returns a list of all the values in the dictionary



LET'S TRY IT

From the Python Shell, enter the following and observe the results.

```
>>> fruit_prices = {'apples': .66, 'pears': .25,
                    'peaches': .74, 'bananas': .49}
>>> fruit_prices['apples']
???
>>> fruit_prices[0]
???
>>> veg_data = [['corn', .25], ['tomatoes', .49], ['peas', .39]]
>>> veg_prices = dict(veg_data)
>>> veg_prices
???
>>> veg_prices['peas']
???
```

Exercise

MCQs

1. A dictionary type in Python is an associative data structure that is accessed by a ________________ rather than an index value.

2. Associative data structures such as the dictionary type in Python are useful for,
   a) accessing elements more intuitively than by use of an indexed data structure
   b) maintaining elements in a particular order

3. Which of the following types can be used as a key in Python dictionaries?
   a) strings
   b) lists
   c) tuples
   d) numerical values

4. Which of the following is a syntactically correct sequence, s, for dynamically creating a dictionary using *dict(s)*.
   a) s = [[1: 'one'], [2: 'two'], [3: 'three']]
   b) s = [[1, 'one'], [2, 'two'], [3, 'three']]
   c) s = {1:'one', 2:'two', 3:'three'}

5. For dictionary *d = {'apples' : 0.66, 'pears' : 1.25, 'bananas' : 0.49},* which of the following correctly updates the price of bananas.
   a) d[2] = 0.52
   b) d[0.49] = 0.52
   c) d['bananas'] = 0.52

MCQs: Answers

1. A dictionary type in Python is an associative data structure that is accessed by a **key value** rather than an index value.

2. Associative data structures such as the dictionary type in Python are useful for,
   - **a) accessing elements more intuitively than by use of an indexed data structure**
   - b) maintaining elements in a particular order

3. Which of the following types can be used as a key in Python dictionaries?
   - **a) strings**
   - b) lists
   - **c) tuples**
   - **d) numerical values**

4. Which of the following is a syntactically correct sequence, s, for dynamically creating a dictionary using *dict(s)*.
   - a) s = [[1: 'one'], [2: 'two'], [3: 'three']]
   - **b) s = [[1, 'one'], [2, 'two'], [3, 'three']]**
   - c) s = {1:'one', 2:'two', 3:'three’}

5. For dictionary *d = {'apples’ : 0.66, 'pears’ : 1.25, 'bananas’ : 0.49},* which of the following correctly updates the price of bananas.
   - a) d[2] = 0.52
   - b) d[0.49] = 0.52
   - **c) d['bananas’] = 0.52**

Thank You

?