3 Days Training on Python3

Day 3 : Module 11

Muhammad Saufy Rohmad

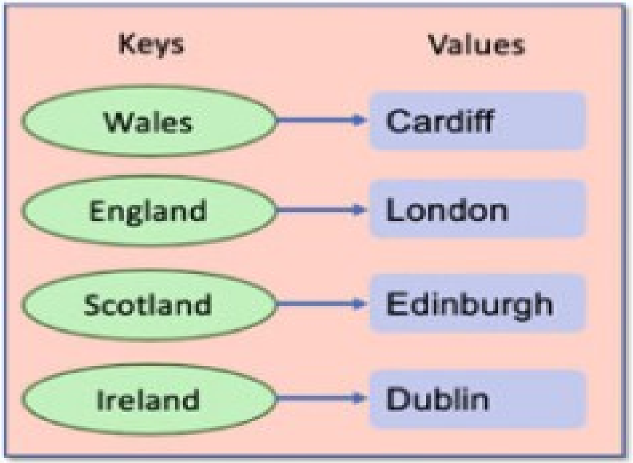
Module 11 (90 minutes)

Objectives

1. Dictionaries
2. ADTs,Queus and Stacks

1. Dictionaries

- A Dictionary is a set of associations between a key and a value that is unordered, changeable (mutable) and indexed.
- Pictorially we might view a Dictionary as shown below for a set of countries and their capital cities.
- Note that in a Dictionary the keys must be unique but the values do not need to be unique.



1.1 Creating Dictionaries

- A Dictionary is created using curly brackets ('{}') where each entry in the dictionary is a key:value pair:

```
cities = {'Wales': 'Cardiff','England': 'London','Scotland': 'Edinburgh','Northern Ireland': 'Belfast', 'Ireland': 'Dublin'}

print(cities)
```

- This creates a dictionary referenced by the variable cities which holds a set of key:value pairs for the Capital cities of the UK and Ireland

1.2 Dictionary Constructor

- The dict() function can be used to create a new dictionary object from an iterable or a sequence of key:value pairs. The signature of this function is:

  dict(**kwarg)

  dict(mapping, **kwarg)

  dict(iterable, **kwarg)

- This is an overloaded function with three version that can take different types of arguments:
  - The first option takes a sequence of key:value pairs.
  - The second takes a mapping and (optionally) a sequence of key:value pairs.
  - The third version takes an iterable of key:value pairs and an optional sequence
  - of key:value pairs.

1.2 Dictionary Constructor(2)

- Example of dict creation

```
# note keys are not strings
dict1 = dict(uk='London', ireland='Dublin', france='Paris')
print('dict1:', dict1)
# key value pairs are tuples
dict2 = dict([('uk', 'London'), ('ireland', 'Dublin'),
('france', 'Paris')])
print('dict2:', dict2)
# key value pairs are lists
dict3 = dict((['uk', 'London'], ['ireland', 'Dublin'],
['france', 'Paris']))
print('dict3:', dict3)
```

1.3 Working with Dictionaries(2)

- You can access the values held in a Dictionary using their associated key

```
print('cities[Wales]:', cities['Wales'])
print('cities.get(Ireland):', cities.get('Ireland'))
```

- Adding new entry

```
cities['France'] = 'Paris'
```

- Changing a Keys Value by reassigning a new value

```
cities['Wales'] = 'Swansea'
print(cities)
```

1.3 Working with Dictionaries(3)

- Removing Entry

```
cities = {'Wales': 'Cardiff', 'England': 'London', 'Scotland':
'Edinburgh','Northern Ireland': 'Belfast','Ireland': 'Dublin'}

print(cities)

cities.popitem() # Deletes 'Ireland' entry

print(cities)

cities.pop('Northern Ireland')

print(cities)

del cities['Scotland']

print(cities)
```

1.3 Working with Dictionaries(4)

- In addition the clear() method empties the dictionary of all entries:

```
cities = {'Wales': 'Cardiff','England':
'London','Scotland': 'Edinburgh','Northern
Ireland': 'Belfast','Ireland': 'Dublin'}

print(cities)

cities.clear()

print(cities)
```

1.4 Iterating Over Keys

- The for loop processes each of the keys in the dictionary in turn

```
for country in cities:
    print(country, end=', ')
    print(cities[country])
```

- There are three methods that allow you to obtain a view onto the contents of a dictionary, these are values(), keys() and items().

```
print(cities.values())
print(cities.keys())
print(cities.items())
```

1.5 Other Operations

- You can check to see if a key is a member of a dictionary

```
print('Wales' in cities)

print('France' not in cities)
```

- you can find out the length of a Dictionary

```
cities = {'Wales': 'Cardiff','England': 'London','Scotland': 'Edinburgh','Northern Ireland': 'Belfast','Ireland': 'Dublin'}

print(len(cities)) # prints 5
```

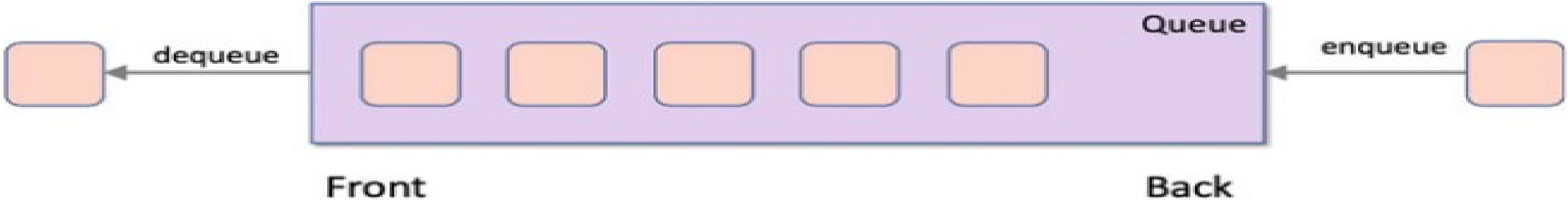
2. ADTs,Queues and Stacks

- There are a number of common data structures that are used within computer programs that you might expect to see within Python’s list of collection or container classes; these include Queues and Stacks
- The Queue and Stack are concrete examples of what are known as Abstract Data Types (or ADTs).
- An Abstract Data Type (or ADT) is a model for a particular type of data, where a data type is defined by its behaviour (or semantics) from the point of view of the user of that data type

2.1 Queus

- Queues are very widely used within Computer Science and in Software Engineering
- There are numerous variations on the basic queue operations but in essence all queues provide the following features.
    - Queue creation.
    - Add an element to the back of the queue (known as enqueuing).
    - Remove an element from the front of the queue (known as dequeuing).
    - Find out the length of the queue.
    - Check to see if the queue is empty.
    - Queues can be of fixed size or variable (growable) in size.

2.1 Queus(2)



- The Python List container can be used as a queue using the existing operations such as append() and pop(), for example:

```
queue = [ ] # Create an empty queue
queue.append('task1')
print('initial queue:', queue)
queue.append('task2')
queue.append('task3')
print('queue after additions:', queue)
element1 = queue.pop(0)
print('element retrieved from queue:', element1)
print('queue after removal', queue)
```

2.1 Queus(3)

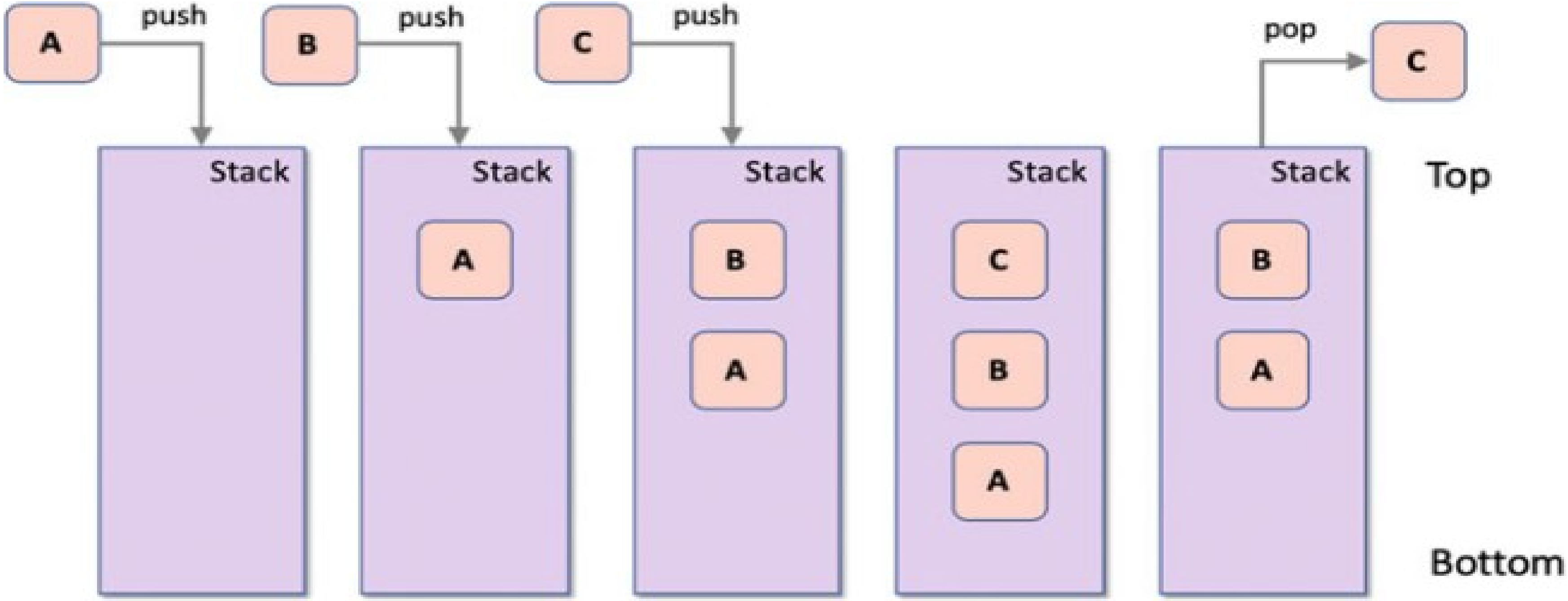
- Using queue class and normal operation

```
queue = Queue()
print('queue.is_empty():', queue.is_empty())
queue.enqueue('task1')
print('len(queue):', len(queue))
queue.enqueue('task2')
queue.enqueue('task3')
print('queue:', queue)
print('queue.peek():', queue.peek())
print('queue.dequeue():', queue.dequeue())
print('queue:', queue)
```

2.2 Stacks

- Stacks are another very widely used ADT within computer science and in software applications. They are often used for evaluating athematical expressions, parsing syntax, for managing intermediate results etc.
- The basic facilities provided by a Stack include:
  - Stack creation.
  - Add an element to the top of the stack (known as pushing onto the stack).
  - Remove an element from the top of the stack (known as popping from the
  - stack).
  - Find out the length of the stack.
  - Check to see if the stack is empty.
  - Stacks can be of fixed size or a variable (growable) stack.

2.2 Stacks(2)


A
push
B
push
C
push
pop
C
Stack
Stack
A
Stack
B
A
Stack
C
B
A
Stack
B
A
Top
Bottom

2.2 Stacks(3)

- A List may initially appear particularly well suited to being used as a Stack as the basic append() and pop() methods can be used to emulate the stack behaviour.

```
stack = [] # create an empty stack
stack.append('task1')
stack.append('task2')
stack.append('task3')
print('stack:', stack)
top_element = stack.pop()
print('top_element:', top_element)
print('stack:', stack)
```