Python Programming

`tuple`, `dict` and solving puzzles

Rathindra Nath Dutta

Senior Research Fellow
Advanced Computing & Microelectronics Unit
Indian Statistical Institute, Kolkata

August 31, 2023

https://www.isical.ac.in/~rathin_r/uploads/PyAI/day3.pdf

tuple

- Stores ordered collection of items
- Tuples are immutable: cannot add/delete/modify items

tuple

- Stores ordered collection of items
- Tuples are immutable: cannot add/delete/modify items

```
x = tuple() # creates an empty tuple
x = () # shorthand
x = (10, '20', 30.1, 10)
```

tuple

- Stores ordered collection of items
- Tuples are immutable: cannot add/delete/modify items

```
x = tuple() # creates an empty tuple
x = () # shorthand
x = (10, '20', 30.1, 10)
```

- Accessing items, slicing etc. can be done similarly

```
print("third element :", x[2])
print("first two elements :", x[:2])
print("last two elements :", x[-2:])
print("joining tuples :", x + x)
print("repeating tuples :", x * 2)
```

tuple

- Stores ordered collection of items
- Tuples are immutable: cannot add/delete/modify items

```
x = tuple() # creates an empty tuple
x = () # shorthand
x = (10, '20', 30.1, 10)
```

- Accessing items, slicing etc. can be done similarly

```
print("third element :", x[2])
print("first two elements :", x[:2])
print("last two elements :", x[-2:])
print("joining tuples :", x + x)
print("repeating tuples :", x * 2)
```

- Conversion between types

```
a = tuple( [10, 20, 30] )
b = list( x )
c = tuple( 'abcd' )
d = list( 'abcd' )
```

set

- Stores unordered collection of distinct hashable[1] objects
- Implements basic mathematical set operations

```
s = set() # s = {} creates a dict
x = [10, 20, 30, 40, 10]
y = [40, 50, 60, 70]
A, B = set(x), set(y)
print(A, B)
print( A.union(B) )
print( A.intersection(B) )
print( A.difference(B) )
A.add(100)
B.remove(40)
...
```

[1] https://docs.python.org/3.8/glossary.html#term-hashable

set

- Stores unordered collection of distinct hashable[1] objects
- Implements basic mathematical set operations

```
s = set() # s = {} creates a dict
x = [10, 20, 30, 40, 10]
y = [40, 50, 60, 70]
A, B = set(x), set(y)
print(A, B)
print( A.union(B) )
print( A.intersection(B) )
print( A.difference(B) )
A.add(100)
B.remove(40)
...
```

- Exercise: report unique items [10, 10, 20, 10, 20, 30, 10, 20, 30, 40]

[1] https://docs.python.org/3.8/glossary.html#term-hashable

dict

- Dictionary is one of the most versatile built-in type in Python
- Stores a collection of $\underline{\langle key, value\rangle \text{ pairs}}$
  Denoted as $\{key1 : value1,\ \ key2 : value2,\ \ \ldots\}$

```
x = {1: 'Python', 2: 'for', 3: 'AI', 4: 'for', 5: 'CUCSE'}
print(x, "is of type", type(x))
```

dict

- Dictionary is one of the most versatile built-in type in Python
- Stores a collection of $\langle key, value \rangle$ pairs
  Denoted as $\{key1 : value1,\ \ key2 : value2,\ \ \ldots\}$

```
x = {1: 'Python', 2: 'for', 3: 'AI', 4: 'for', 5: 'CUCSE'}
print(x, "is of type", type(x))
```

- Values may repeat, but keys must be distinct and hashable

dict

- Dictionary is one of the most versatile built-in type in Python
- Stores a collection of $\langle key, value\rangle$ pairs
  Denoted as $\{key1 : value1,\ \ key2 : value2,\ \ \ldots\}$

```
x = {1: 'Python', 2: 'for', 3: 'AI', 4: 'for', 5: 'CUCSE'}
print(x, "is of type", type(x))
```

- Values may repeat, but keys must be distinct and hashable

```
x = {1: 'Python', 'two': [1, 2.1, 'abc'], 3.31: 1234 }
print(x)
print(x[1], x['two'], x[3.31]) # indexed by the keys
x[1] = 1111 # updating an item
x['two'].append(123) # modifing a member list
del x[3.31] # deleting an index
print(x)
```

dict

- Dictionary is one of the most versatile built-in type in Python
- Stores a collection of $\langle key, value \rangle$ pairs
  Denoted as $\{key1 : value1, \;\; key2 : value2, \;\; \ldots\}$

```
x = {1: 'Python', 2: 'for', 3: 'AI', 4: 'for', 5: 'CUCSE'}
print(x, "is of type", type(x))
```

- Values may repeat, but keys must be distinct and hashable

```
x = {1: 'Python', 'two': [1, 2.1, 'abc'], 3.31: 1234 }
print(x)
print(x[1], x['two'], x[3.31]) # indexed by the keys
x[1] = 1111 # updating an item
x['two'].append(123) # modifing a member list
del x[3.31] # deleting an index
print(x)
```

- Explore: `len(x)`, `key in x`, `key not in x`, `x.clear()` etc.

Iterating a `dict`

Check the returned values of: `x.values()`, `x.keys()`, `x.items()`

Iterating a dict

Check the returned values of: `x.values()`, `x.keys()`, `x.items()`

```
quantity = {"apple": 1, "orange": 2, "eggs": 3}
count = 0
for v in quantity.values():
    count += v
print(count)
```

Iterating a `dict`

Check the returned values of: `x.values()`, `x.keys()`, `x.items()`

```
quantity = {"apple": 1, "orange": 2, "eggs": 3}
count = 0
for k in quantity.keys():
    count += quantity[k]
print(count)
```

Iterating a `dict`

Check the returned values of: `x.values()`, `x.keys()`, `x.items()`

```
quantity = {"apple": 1, "orange": 2, "eggs": 3}
price = {"apple": 50, "orange": 20, "eggs": 10}
count = 0
total = 0
for k,v in quantity.items():
    count += v # quantity[k]
    total += price[k] * v
print(count, total)
```

Solving the Water Jug Problem

```python
class State:
    goal = None  # common for all states
    capacity1 = None
    capacity2 = None
    def __init__(self, filled1, filled2):
        self.filled1 = filled1
        self.filled2 = filled2


State.capacity1 = 40
State.capacity2 = 70
State.goal = 10

start_state = State(0, 0)
```

Solving the Water Jug Problem

```
class State:
    goal = None  # common for all states
    capacity1 = None
    capacity2 = None
    def __init__(self, filled1, filled2):
        self.filled1 = filled1
        self.filled2 = filled2


State.capacity1 = 40
State.capacity2 = 70
State.goal = 10

start_state = State(0, 0)

print(start_state) # prints object id in hex
```

Solving the Water Jug Problem

Printable string representation of an object can be obtained by overriding: `__str__()` or `__repr__()` methods

```
class State:
    ...
    def __repr__(self): # must return a string
        return f'<jug1: {self.filled1}/{State.capacity1},
        ↪  jug2: {self.filled2}/{State.capacity2},
        ↪  goal: {State.goal}>' # formatted string
    def __str__(self):
        return self.__repr__() # return the same

State.capacity1 = 40
State.capacity2 = 70
State.goal = 10
start_state = State(0, 0)
print(start_state)
```

Solving the Water Jug Problem

```
class State:
    ...

State.capacity1 = 40
State.capacity2 = 70
State.goal = 10

start_state = State(0, 0)
s1 = State(0, 0)
print(start_state)
print(s1)

print(start_state is s1) # False
print(start_state == s1) # False
```

Solving the Water Jug Problem

Implement the rich comparison[1] method(s) as per the requirements

`x == y` calls `x.__eq__(y)`

```
class State:
    ...
    def __eq__(self, other): # for comparing two objects
        return self.filled1 == other.filled1 and
        ↪  self.filled2 == other.filled2

State.capacity1 = 40
State.capacity2 = 70
State.goal = 10
start_state = State(0, 0)
s1 = State(0, 0)
print(start_state is s1) # False
print(start_state == s1) # True
```

[1] https://docs.python.org/3/reference/datamodel.html#object.__lt__

Solving the Water Jug Problem

```
class State:
    ...
    def make_move(self):
        next_states = []
        if self.filled1 < State.capacity1: # pour into jug1
            s = State(State.capacity1, self.filled2)
            next_states.append( s )
```

Solving the Water Jug Problem

```
class State:
    ...
    def make_move(self):
        next_states = []
        if self.filled1 < State.capacity1: # pour into jug1
            s = State(State.capacity1, self.filled2)
            next_states.append( s )
            remaining_capacity1 = State.capacity1 - self.filled1
            if remaining_capacity1 >= self.filled2: # empty jug2
            ↪  into jug1
                s = State(self.filled1 + self.filled2, 0)
                next_states.append( s )
```

Solving the Water Jug Problem

```
class State:
    ...
    def make_move(self):
        next_states = []
        if self.filled1 < State.capacity1: # pour into jug1
            s = State(State.capacity1, self.filled2)
            next_states.append( s )
            remaining_capacity1 = State.capacity1 - self.filled1
            if remaining_capacity1 >= self.filled2: # empty jug2
            ↪  into jug1
                s = State(self.filled1 + self.filled2, 0)
                next_states.append( s )
            else: # pour as much as possible from jug2 to jug1
                s = State(State.capacity1, self.filled2 -
                ↪  remaining_capacity1)
                next_states.append( s )
```

Solving the Water Jug Problem

```
class State:
    ...
    def make_move(self):
        next_states = []
        if self.filled1 < State.capacity1: # pour into jug1
            s = State(State.capacity1, self.filled2)
            next_states.append( s )
            remaining_capacity1 = State.capacity1 - self.filled1
            if remaining_capacity1 >= self.filled2: # empty jug2
            ↪  into jug1
                s = State(self.filled1 + self.filled2, 0)
                next_states.append( s )
            else: # pour as much as possible from jug2 to jug1
                s = State(State.capacity1, self.filled2 -
                ↪  remaining_capacity1)
                next_states.append( s )
        if self.filled2 < State.capacity2: # pour into jug2
            ... # symmetric cases
        return next_states
```

Solving the Water Jug Problem

```
class State:
    ...
    def is_final(self):
        if self.filled1 == State.goal or self.filled2 == State.goal:
            return True
        return False
```

Solving the Water Jug Problem

Complete class: water_jug.py

```
State.capacity1 = 40
State.capacity2 = 70
State.goal = 10
start_state = State(0, 0)

next_states = start_state.make_move()
print( next_states )
```

Solving the Water Jug Problem

```
def find_sol_BFS(source):
    OPEN = Queue()
    CLOSED = []

    OPEN.enqueue(source)
    CLOSED.append(source) # visited
    while( not OPEN.is_empty() ):
        u = OPEN.dequeue()
        # print(u, end=', ') # process node u
        if u.is_final():
            print("solution found")
            break
        for v in u.make_move(): # next states
            if v not in CLOSED:
                OPEN.enqueue(v)
                CLOSED.append(v) # visited
    else:
        print("no sol exists")
```

Solving the Water Jug Problem

- How to print the solution path?

Solving the Water Jug Problem

- How to print the solution path?
- Back trace from the goal state to the start state

Solving the Water Jug Problem

- How to print the solution path?
- Back trace from the goal state to the start state
- Keep a parent reference in the state, parent of start state is `None`

Solving the Water Jug Problem

- How to print the solution path?
- Back trace from the goal state to the start state
- Keep a parent reference in the state, parent of start state is `None`
- Print the trace (in reverse) when goal is reached

Solving the Water Jug Problem

- How to print the solution path?
- Back trace from the goal state to the start state
- Keep a parent reference in the state, parent of start state is `None`
- Print the trace (in reverse) when goal is reached

```
def print_sol_trace(state):
    trace = [state]
    while True:
        if state.parent is None:
            print(trace)
            break
        trace.insert(0, state.parent) # add in the front
        state = state.parent
```

Solving the Water Jug Problem

- How to print the solution path?
- Back trace from the goal state to the start state
- Keep a parent reference in the state, parent of start state is `None`
- Print the trace (in reverse) when goal is reached

```
def print_sol_trace(state):
    trace = [state]
    while True:
        if state.parent is None:
            print(trace)
            break
        trace.insert(0, state.parent) # add in the front
        state = state.parent
```

Final code: water_jug_ver2.py and my_queue.py

Practice Problems

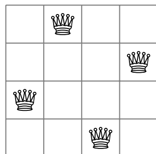
- River crossing puzzles
  - Missionaries and cannibals problem

Practice Problems

- River crossing puzzles
  - Missionaries and cannibals problem
  - Wolf, goat and cabbage problem

Practice Problems

- River crossing puzzles
  - Missionaries and cannibals problem
  - Wolf, goat and cabbage problem
- n-Queens problem



Practice Problems

- River crossing puzzles
  - Missionaries and cannibals problem
  - Wolf, goat and cabbage problem
- n-Queens problem
- n-puzzle problem

1	4	2
6		5
7	3	8

Practice Problems

- River crossing puzzles
  - Missionaries and cannibals problem
  - Wolf, goat and cabbage problem
- n-Queens problem
- n-puzzle problem