Chapter 2

Data Structures

We define a **data structure** as a specific way to group and manage the information, such that we can efficiently use the data. Opposite to the simple variables, a data structure is an abstract data type that involves a high level of abstraction, and therefore a tight relation with *OOP*. We will show the Python implementation of every data structure according to its conceptual model. Each data structure depends on the problem's context and design, and the expected efficiency of our algorithm. In conclusion, choosing the right data structure impacts directly on the outcome of any software development project.

In Python, we could create a simple data structure by using an empty object without methods and add the attributes along with our program. However, using empty classes is not recommended, because:

- *i)* it requires a lot of memory to keep tracked all the potentially new attributes, names, and values.
- *ii)* it decreases the maintainability of the code.
- *iii)* it is an overkill solution.

The example below shows the use of the `pass` sentence to let the class empty, which corresponds to a null operation. We commonly use the `pass` sentence when we expect the method to be defined later. Once we create the object, we can add more attributes.

```
1  # We create an empty class
2  class Video:
3      pass
4
5  vid = Video()
```

```
6
7  # We add new attributes
8  vid.ext = 'avi'
9  vid.size = '1024'
10
11 print(vid.ext, vid.size)
```

```
avi 1024
```

We can also create a class only with few attributes, but still without methods. Python allows us to add new attributes to our class on the fly.

```
1  # We create a class with some attributes
2  class Image:
3
4      def __init__(self):
5          self.ext = ''
6          self.size = ''
7          self.data = ''
8
9
10 # Create an instance of the Image class
11 img = Image()
12 img.ext = 'bmp'
13 img.size = '8'
14 img.data = [255, 255, 255, 200, 34, 35]
15
16 # We add this new attribute dynamically
17 img.ids = 20
18
19 print(img.ext, img.size, img.data, img.ids)
```

```
bmp 8 [255, 255, 255, 200, 34, 35] 20
```

Fortunately, Python has many built-in data structures that let us manage data efficiently, such as: `list`, `tuples`, `dictionaries`, `sets`, `stacks`, and `queues`.

2.1 Array-Based Data Structures

In this section, we will review a group of data structures based on the sequential order of their elements. These kinds of structures are indexed through `seq[index]`. Python uses an index format that goes from 0 to $n-1$, where n is the number of elements in the sequence. Examples of this type of structures are: `tuple` and `list`.

Tuples

Tuples are useful for handling ordered data. We can get a particular element inside the tuple by using its index:

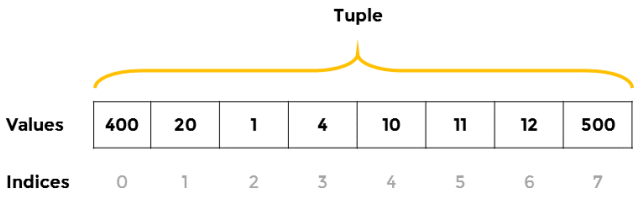

Figure 2.1: Diagram of indexing on tuples. Each cell contains a value of the tuple that could be referenced using its index. In Python, indices go from 0 until $n-1$, where the tuple has length n.

Tuples can handle various kind of data types. We can create a tuple using the tuple constructor as follows: `tuple(`$element_0, element_1, \ldots, element_{n-1}$`)`. We can create a empty tuple using `tuple()` without arguments: `a = tuple()`. We can also create a tuple by directly adding the tuple elements:

```
1 b = (0, 1, 2)
2 print(b[0], b[1])
```

```
0 1
```

A tuple can handle various data types. The parentheses are not mandatory during its creation:

```
1 c = 0, 'message'
2 print(c[0], c[1])
```

```
0 message
```

We can also add any object to the tuple:

```
1 teacher = ('Christian', '23112436-0', 2)
2 video = ('data-structures.avi', 1024, 'mp4')
3 entry = (1, teacher, video)
4 print(entry)
```

```
(1, ('Christian', '23112436-0', 2), ('data-structures.avi', 1024, 'mp4'))
```

Tuples are **immutable**, *i.e*, once we create a tuple, it is not possible to add, remove or change elements of the tuple. This immutability allows us to use tuples as a key value in hashing-based data structures, such as dictionaries. In the next example, we create a tuple with three elements: an instance of the class `Image`, a `string`, and a `float`. Then, we attempt to change the element in position 0 by a string. We can see that this attempt raise a `TypeError` exception:

```
1 a = ('this is' , 'a tuple', 'of strings')
2 a[1] = 'new data'

Traceback (most recent call last):
  File "05_tuple_inmutable.py", line 2, in <module>
    a[1] = 'new data'
TypeError: 'tuple' object does not support item assignment
```

We can map tuples into a set of individual variables. For example, if a function returns a tuple with several values, the tuple can be assigned separately to a set of individual variables. The code below shows an example, the function `compute_geometry()` receives as input the sides a and b of a quadrilateral and returns a set of geometric measures:

```
1  def compute_geometry(a, b):
2      area = a * b
3      perimeter = (2 * a) + (2 * b)
4      mpa = a / 2
5      mpb = b / 2
6
7      return (area, perimeter, mpa, mpb)
8
9  data = compute_geometry(20.0, 10.0)
10 print('1: {0}'.format(data))
11
12 a = data[0]
13 print('2: {0}'.format(a))
14
15 # Here we unpack the values into independent variables contained
16 # in the tuple
17 a, p, mpa, mpb = data
```

```
18 print('3: {0}, {1}, {2}, {3}'.format(a, p, mpa, mpb))
19 a, p, mpa, mpb = compute_geometry(20.0, 10.0)
20 print('4: {0}, {1}, {2}, {3}'.format(a, p, mpa, mpb))
```

```
1: (200.0, 60.0, 10.0, 5.0)
2: 200.0
3: 200.0, 60.0, 10.0, 5.0
4: 200.0, 60.0, 10.0, 5.0
```

We can use slice notation to select a section of the tuple. In this notation, indexes do not correspond directly to the element positions in the sequence, but they work as boundaries to indicate `sequence[start:stop:steps]`. As a default, `steps` $= 1$. Figure 2.2 shows an example.

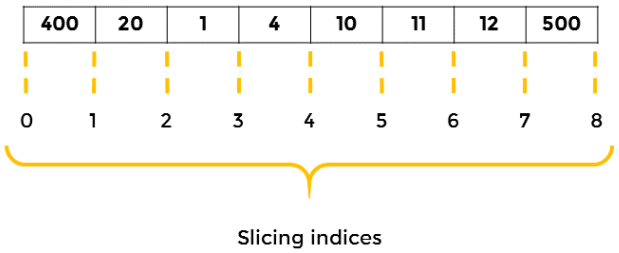

Figure 2.2: Slicing example. Python allows selecting a portion of a tuple or a list using the slice notation. Opposite to a single indexing, slicing start at 0 until n, where n is the length of the sequence.

```
1  data = (400, 20, 1, 4, 10, 11, 12, 500)
2  a = data[1:3]
3  print('1: {0}'.format(a))
4  a = data[3:]
5  print('2: {0}'.format(a))
6  a = data[:5]
7  print('3: {0}'.format(a))
8  a = data[2::2]
9  print('4: {0}'.format(a))
10 #We can revert a sequence:
11 a = data[::-1]
12 print('5: {0}'.format(a))
```

```
1: (20, 1)
2: (4, 10, 11, 12, 500)
3: (400, 20, 1, 4, 10)
4: (1, 10, 12)
5: (500, 12, 11, 10, 4, 1, 20, 400)
```

Named Tuples

Named Tuples let us define a name for each position of the data. They are useful to group elements. First, we require to import the module **namedtuple** from library **collections**. Then, we need to define an object with the tuple attribute names:

```
from collections import namedtuple

# name of tuple type (defined by user) and tuple attributes
Register = namedtuple('Register', 'ID_NUMBER name age')
c1 = Register('13427974-5', 'Christian', 20)
c2 = Register('23066987-2', 'Dante', 5)
print(c1.ID_NUMBER)
print(c2.ID_NUMBER)
```

```
13427974-5
23066987-2
```

Functions can also return Named Tuples:

```
from collections import namedtuple

def compute_geometry(a, b):
    Features = namedtuple('Geometrical', 'area perimeter mpa mpb')
    area = a*b
    perimeter = (2*a) + (2*b)
    mpa = a/2
    mpb = b/2
    return Features(area, perimeter, mpa, mpb)

data = compute_geometry(20.0, 10.0)
print(data.area)
```

```
200.0
```

Lists

This data structure allows us to manage multiple instances of the same type of object, although, they are not limited to combine various type of object classes. Lists are sequential data structures, sorted according to the order we add its elements. Opposite to tuples, lists are **mutable**, *i.e*, their content can dynamically change after their creation.

We must avoid using lists to collect various attributes of an object or using them as vectors in C++, for example as a histogram of words frequency `['python', 20, 'language', 16]`. This way requires an algorithm to access the data inside the list that makes hard use it. In these cases, we must prefer another data structure such as hashing-based data structures, `NamedTuples`, or simply a `dictionary`.

```
# An empty list. We add elements one-by-one
# In this case we add tuples
le = []
le.append((2015, 3, 14))
le.append((2015, 4, 18))
print(le)

# We can also explicitly assign values during creation
l = [1, 'string', 20.5, (23, 45)]
print(l)

# We can retrieve an element using their index
print(l[1])
```

```
[(2015, 3, 14), (2015, 4, 18)]
[1, 'string', 20.5, (23, 45)]
string
```

A useful lists method is `extend()` that allows us to add a complete list to other list already created.

```
# We create a list with 3 elements
songs = ['Addicted to pain', 'Ghost love score', 'As I am']
print(songs)

# Then, we add the list "songs" to the list "new_songs"
```

```
new_songs = ['Elevate', 'Shine']
songs.extend(new_songs)
print(songs)
```

```
['Addicted to pain', 'Ghost love score', 'As I am']
['Addicted to pain', 'Ghost love score', 'As I am', 'Elevate', 'Shine']
```

We can also insert elements at specific positions within the list using the method `insert(position, element)`.

```
# We create a list with 3 elements
songs = ['Addicted to pain', 'Ghost love score', 'As I am']
print(songs)

# Then, we insert a new songs at the position 1
songs.insert(1, 'Sober')
print(songs)
```

```
['Addicted to pain', 'Ghost love score', 'As I am']
['Addicted to pain', 'Sober', 'Ghost love score', 'As I am']
```

In addition, we can ask for an element using the index or retrieve a portion of the list using *slicing* notation. Here we show some examples:

```
# We can take a slice
numbers = [6,7,2,4,10,20,25]
print(numbers[2:6])
```

```
[2, 4, 10, 20]
```

```
# We can pick a portion until the end
print(numbers[2::])
```

```
[2, 4, 10, 20, 25]
```

```
# We can take a slice from the beginning until a specific position
print(numbers[:5])
```

```
[6, 7, 2, 4, 10]
```

```
# We can also change the number of steps
print(numbers[:5:2])
```

```
[6, 2, 10]
```

```
# We can revert a list
print(numbers[::-1])
```

```
[25, 20, 10, 4, 2, 7, 6]
```

Lists can be sorted using the method `sort()`. This method sorts the list in place *i.e*, does not return any value.

```
# We create the list with seven numbers
numbers = [6, 7, 2, 4, 10, 20, 25]
print(numbers)

# Ascendence. Note that variable a do not receive
# any value from assignation.
a = numbers.sort()
print(numbers, a)

# Descendent
numbers.sort(reverse=True)
print(numbers)
```

```
[6, 7, 2, 4, 10, 20, 25]
[2, 4, 6, 7, 10, 20, 25] None
[25, 20, 10, 7, 6, 4, 2]
```

Lists are optimized to be flexible and easy to manage. They are easy to use within `for` loops. Note that we avoid using `id` as a variable because it is a reserved word in Python language.

```
class Piece:
    # Avoid using id as variable because it is a reserved word
    pid = 0

    def __init__(self, piece):
        Piece.pid += 1
```

```
7           self.pid = Piece.pid
8           self.type = piece
9
10  pieces = []
11  pieces.append(Piece('Bishop'))
12  pieces.append(Piece('Pawn'))
13  pieces.append(Piece('King'))
14  pieces.append(Piece('Queen'))
15
16  for piece in pieces:
17      print('pid: {0} - types of piece: {1}'.format(piece.pid, piece.type))
```

```
pid: 1 - types of piece: Bishop
pid: 2 - types of piece: Pawn
pid: 3 - types of piece: King
pid: 4 - types of piece: Queen
```

Stacks

Stacks are a data structures that manage the elements using the **Last-in First-out** (LIFO) principle. When we add elements, they are located on top of the stack. When we remove elements from it, we take the most recently added element. The Figure 2.3 shows an analogy between stacks and a pile of clean dishes. The last added dish will be the first dish to be used.

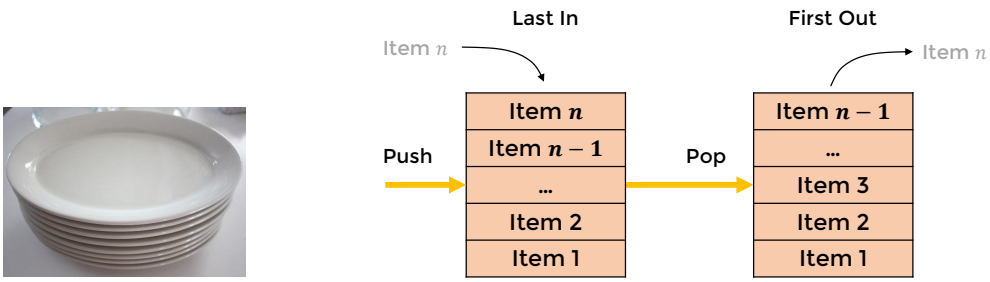

Figure 2.3: Here we show the analogy between `stacks` and a pile of dishes. The `push()` method add an element to the top of the pile. The `pop()` method let us to get the last element added to the stack.

Stacks have two main methods: `push()` and `pop()`. The `push()` method allows us to add an element to the end of the stack and the `pop()` let us to get the top element in the stack. In Python, the stacks are built-in as `Lists`.

There are also more methods, such as: `top()`, `is_empty()`, `len()`. Figure 2.4 includes a brief description and comparison of the other methods included in this data structure.

Basics Methods for Stacks	Python Implementation	Description
Stack.push(item)	List.append(item)	Add sequentialy a new item to the stack
Stack.pop()	List.pop()	Returns and removes the last item added to the stack
Stack.top()	List[-1]	Return the last item added to stack without remove it
len(Stack)	len(List)	Return the total number of items in the stack
Stack.is_empty()	len(List) == 0	Verify whether the stack is empty or not

Figure 2.4: Summary of most used methods of the stack data structure and its equivalence in Python.

Methods described in Figure 2.4 work as follows:

```
1  # Create an empty Stack. In Python Stacks are built-in as Lists.
2  stack = []
3
4  # push() method
5  stack.append(1)
6  stack.append(10)
7  stack.append(12)
8
9  print(stack)
10
11 # pop() method
12 stack.pop()
13 print('pop(): {0}'.format(stack))
14
15 # top() method. Lists does not have a this method implemented directly.
16 #We can have the same behaviour indexing the last element in the Stack.
17 stack.append(25)
18 print('top(): {0}'.format(stack[-1]))
19
20 # len()
21 print('The stack have {0} elements'.format(len(stack)))
22
```

```
23 # is_empty() method. In Python we verify the status of the stack
24 #checking if it has elements.
25 stack  = []
26 if len(stack) == 0:
27     print('The stack is empty :(')
```

```
[1, 10, 12]
pop(): [1, 10]
top(): 25
The stack have 3 elements
The stack is empty :(
```

A practical example of stacks is the back button of web browsers. When we are browsing the internet, each time we visit an URL, the browser add the link to a stack. Then, we can recover the last visited URL when we click on the back button.

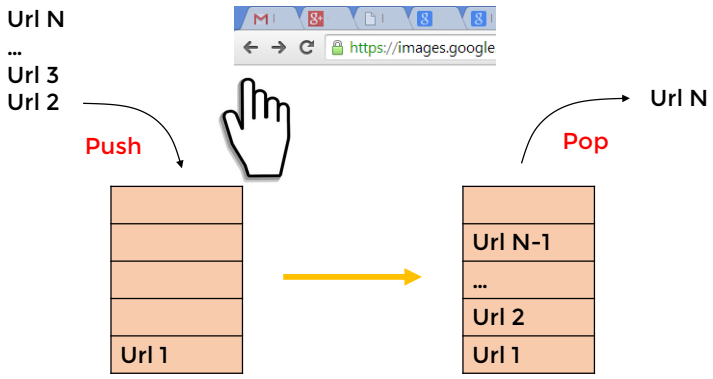

Figure 2.5: An example of using Stacks in a web browser. We can recover the last visited URL every time we press the back button of the browser.

```
1 class Browser:
2
3     def __init__(self, current_url='http://www.google.com'):
4         self.__urls_stack = []
5         self.__current_url = current_url
6
7     def __load_url(self, url):
8         self.__current_url = url
```

```
9           print('loading url: {0}'.format(url))
10
11      def go(self, url):
12          self.__urls_stack.append(self.__current_url)
13          print('go ->', end=' ')
14          self.__load_url(url)
15
16      def back(self):
17          last_url = self.__urls_stack.pop()
18          print('back->', end=' ')
19          self.__load_url(last_url)
20
21      def show_current_url(self):
22          print('Current URL: {0}'.format(self.__current_url))
23
24
25  if __name__ == '__main__':
26      chrome = Browser()
27      chrome.go('http://www.uc.cl')
28      chrome.go('http://www.uc.cl/en/courses')
29      chrome.go('http://www.uc.cl/es/doctorado')
30
31      chrome.show_current_url()
32      chrome.back()
33      chrome.show_current_url()
```

```
go -> loading url: http://www.uc.cl
go -> loading url: http://www.uc.cl/en/courses
go -> loading url: http://www.uc.cl/es/doctorado
Current URL: http://www.uc.cl/es/doctorado
back-> loading url: http://www.uc.cl/en/courses
Current URL: http://www.uc.cl/en/courses
```

Another practical example of stacks is sequence reversion. We show a simple implementation of this task in the next example:

```
1  class Text:
```

```
2       def __init__(self):
3           self.stack = []
4
5       def read_file(self, filename):
6           print('input:')
7
8           with open(filename) as fid:
9               for line in fid:
10                  print(line.strip())
11                  self.stack.append(line.strip())
12
13          print()
14          fid.closed
15
16      def reverse_lines(self):
17          print('output:')
18
19          while len(self.stack) > 0:
20              print(self.stack.pop())
21
22
23  if __name__ == '__main__':
24      t = Text()
25      t.read_file('stacks_text.txt')
26      t.reverse_lines()
```

```
input:
he friend who can be silent with us in a moment of despair or confusion,
who can stay with us in an hour of grief and bereavement,
who can tolerate not knowing... not healing, not curing...
that is a friend who cares.

output:
that is a friend who cares.
who can tolerate not knowing... not healing, not curing...
who can stay with us in an hour of grief and bereavement,
```

```
he friend who can be silent with us in a moment of despair or confusion,
```

Queues

This data structure is an abstract data type that lets us collect objects sequentially following the rule **First–in, First–out** (FIFO). When we add elements, we place them at the end of the queue. When we remove elements from it, we take the oldest element in the queue. Various examples fit with the queue model, such as the line in a supermarket or incoming calls in a call center.

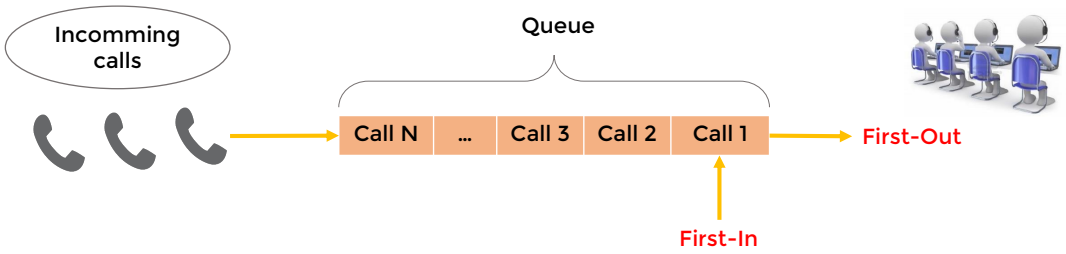

Figure 2.6: A practical example of queues is a call center, where incoming requests wait until an operator can pick the first call added to the queue.

In Python, lists do not work efficiently as queues. Fortunately, the `collections` library includes the `deque` module that is an efficient implementation for data structures based on the FIFO model. This module manages single and double ended queues, while executes all operations efficiently and memory safe. The `deque` module guarantees that the memory access from both sides of the queue is $O(1)$. Although lists support similar operations and methods like queues, they are optimized to perform operations in fixed sized sequences. For example, operations that change the length or position of the element in the sequences, such as `pop(0)` or `insert(0,v)`, involve a cost to update de memory registers of $O(n)$. Queues have two primary methods: `enqueue()` that allows us to add elements to the queue; and `dequeue()`, that returns and removes the first element in the queue. Figure 2.7 includes a brief summary of other methods and operations of this data structure.

```
1 # The collection library includes the deque module that manages single queues
2 # or double ended queues efficiently
3 from collections import deque
4
5 # An empty queue
6 q = deque()
7
```

Basic Methods in Queues	Python Implementation	Description
Queue.enqueue(item)	deque.append(item)	Add an ítem to the queue
Queue.dequeue()	deque.popleft()	Returns and remove the first item in the queue
Queue.first()	deque[1]	Returns the first ítem in the queue without remove it
len(Queue)	len(deque)	Returns the total number of elements in the queue
Queue.is_empty()	len(deque) == 0	Verify whether the queue is empty or not

Figure 2.7: Summary of used methods and operations in queues.

```
8  # We add some elements to the queue using append method, similar to lists.
9  q.append('orange')
10 q.append('apple')
11 q.append('pear')
12
13 # Print the queue
14 print(q)
15
16 # We extract the first element and print again the queue
17 print('Remove the item: {}'.format(q.popleft()))
18 print(q)
19
20 # Add a new element to the queue
21 q.append('strawberry')
22
23 # Get the first element
24 print('The first item is {}'.format(q[0]))
25
26 # len()
27 print('The queue have {0} elements'.format(len(q)))
28
29 # is_empty(). We first remove all the items in the queue using clear() method
30 # and the we check the number of elements.
31 q.clear()
32 if len(q) == 0:
33     print('The queue is empty')
```

```
deque(['orange', 'apple', 'pear'])
Remove the item: orange
deque(['apple', 'pear'])
The first item is pear
The queue have 3 elements
The queue is empty
```

The code below shows a practical example of this data structure applied to vehicle inspection garages:

```
1  from collections import deque
2  from random import choice, randrange
3
4
5  class Vehicle:
6
7      # This class models the vehicles that upcoming to the plant and the
8      # average time spent during the test
9      tp = {'motorcycle': 10, 'car': 25, 'suv': 30}
10
11     def __init__(self):
12         self.vehicle_type = choice(list(Vehicle.tp))
13         self._testing_time = Vehicle.tp[self.vehicle_type]
14
15     @property
16     def testing_time(self):
17         return self._testing_time
18
19     @testing_time.setter
20     def testing_time(self, new_time):
21         self._testing_time = new_time
22
23     def show_type(self):
24         print('Testing: {0}'.format(self.vehicle_type))
25
26
27 class Plant:
```

```
28
29      # This class models the test plant
30
31      def __init__(self, vehicles_per_hour):
32          self.test_ratio = vehicles_per_hour
33          self.current_task = None
34          self.testing_time = 0
35
36      def busy(self):
37          return self.current_task != None
38
39      def next_vehicle(self, vehicle):
40          self.current_task = vehicle
41          self.testing_time = self.current_task.testing_time
42          self.current_task.show_type()
43
44      def tick(self):
45          if self.current_task != None:
46              self.testing_time = self.testing_time - 1
47              if self.testing_time <= 0:
48                  self.current_task = None
49
50
51  def arrive_new_car():
52      # This function manage randomly if arrives a new vehicle
53      num = randrange(1, 201)
54      return num == 200
55
56
57  def testing():
58
59      # This function manage the testing process
60
61      # We createa a Plant with capacity of 5 vehicles/hour
62      plant = Plant(5)
```

```
63
64      # Then, we create an empty queue
65      q = deque()
66
67      # Waiting time
68      time_list = []
69
70      # We model arriving vehicles randomly
71      for instant in range(1000):
72
73          if arrive_new_car():
74              v = Vehicle()
75              q.append(v)
76
77          if (not plant.busy()) and (len(q) > 0):
78              # We get the next vehicle in the queue
79              next_vehicle = q.popleft()
80              time_list.append(next_vehicle.testing_time)
81              plant.next_vehicle(next_vehicle)
82
83          # We make time pass by one tick
84          plant.tick()
85
86      average_time = sum(time_list) / len(time_list)
87      print(
88      'Average waiting time: {0:6.2f} min, {1} vehicles queued.'.format(
89          average_time,
90          len(time_list))
91          )
92
93
94  if __name__ == '__main__':
95      testing()
```

```
Testing: suv
Testing: motorcycle
```

```
Testing: motorcycle
Testing: motorcycle
Testing: suv
Testing: motorcycle
Testing: suv
Average waiting time:  18.57 min, 7 vehicles queued.
```

Double Ended Queue (Deque)

The double ended queue data structure is the general case of stack and queue, which allows us to add and remove items from both sides of the structure. This flexibility is useful for modeling real problems where the entities have a different frequency for arrival and attention. This difference originates that some objects leave the queue early. A real example is a call center that receives incoming calls with different priorities and others calls finish abruptly.

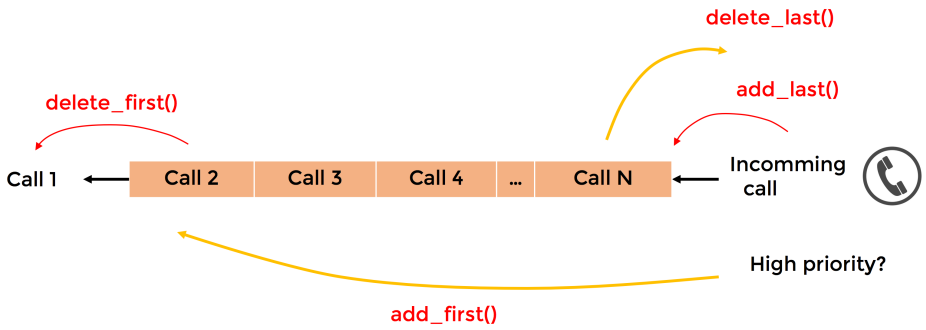

Figure 2.8: Example of using a double ended queues.

Similar to queues, we create a *deque* structure using the **deque** module included in the **collections** library. The table in Figure 2.9 shows a summary of the basic methods and operations of a *deque*.

```
1  from collections import deque
2
3  # We create an empty deque and item by item
4  d = deque()
5  d.append('r')
6  d.append('a')
7  d.append('d')
8  d.append('a')
9  d.append('r')
10 d.append('e')
```

Collection.deque	Python Implementation	Description
Deque.add_first(item)	Deque.appendleft(item)	Add an item at the begining of the deque
Deque.add_last(item)	Deque.append(item)	Add an item at the end of the deque
Deque.delete_first()	Deque.popleft()	Returns and removes the first ítem
Deque.delete_last()	Deque.pop()	Returns and remove the last item
Deque.first()	Deque[0]	Returns the first item without extract it
Deque.last()	Deque[-1]	Returns the last item without extract it
len(Deque)	len(Deque)	Returns the total number of ítems in the deque
Deque.is_empty()	len(Deque) == 0	Verify whether the deque is empty or not
	Deque[j]	Access to the item j
	Deque[j] = value	Modify the item j
	Deque.clear()	Delete all the ítems
	Deque.rotate(k)	K step circular scrolling
	Deque.remove(e)	Remove the first item that match to e
	Deque.count(e)	Counts the number of items that match to e

Figure 2.9: Brief summary of the basic `deque` methods in python.

```
11 d.append('s')
12
13 print(d)
14 print(len(d))

   deque(['r', 'a', 'd', 'a', 'r', 'e', 's'])
   7

1  # Next, we check the first and the last items
2  print(d[0], d[-1])

   r s

1  # Then, we rotate the deque in k=3 step
2  d.rotate(3)
3  print(d)

   deque(['r', 'e', 's', 'r', 'a', 'd', 'a'])

1  # Finally, we extract the first and the last items
2  first = d.popleft()
3  last = d.pop()
4
```

```
print(first, last)
print(d)
```

```
r a
deque(['e', 's', 'r', 'a', 'd'])
```

Next, a simple example of the use of a *deque* for palindrome detection. The word is saved in a *deque*, while we iteratively remove and compare the first and the last characters.

```
from collections import deque


class Word:

    def __init__(self, word=None):
        self.word = word
        self.characters = deque(self.word)

    def is_palindrome(self):
        if len(self.characters) > 1:
            return self.characters.popleft() == self.characters.pop() \
             and Word(''.join(self.characters)).is_palindrome()
        else:
            return True

p1 = Word("radar")
p2 = Word("level")
p3 = Word("structure")

print(p1.is_palindrome())
print(p2.is_palindrome())
print(p3.is_palindrome())
```

```
True
True
False
```

Note: In Python, if we want to check whether a word is a palindrome or not, we only need to compare word == word[::-1].

Dictionaries

A **dictionary** is a data structure based on **key–value** associations. This relation allows us to use the **key** to efficiently search an item because it points to the memory address where its associated **value** is located.

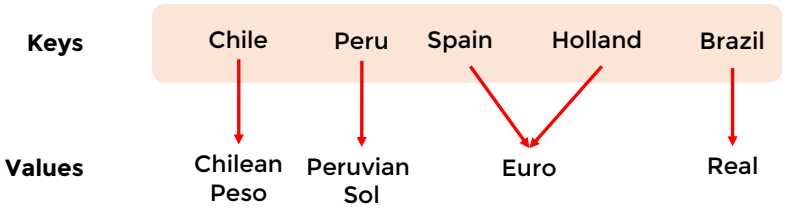

Figure 2.10: Example of a dictionary. Every key is unique in the dictionary.

In Python, an empty dictionary is created by braces `{}` or `dict()`. We must specify the key and value using colon :, *i.e*, `{key1: value1, key2: value2, ...}`. The key has to be an inmutable object: `int`, `str`, `tuple`, etc. We can access the value using the key within **brackets**. The next example shows how we create a dictionary:

```
1  dogs = {'bc': 'border-collie', 'lr': 'labrador retriever', 'pg': 'pug'}
2  telephones = {23545344: 'John', 23545340: 'Trinity', 23545342: 'Taylor'}
3  tuples = {('23545344', 0): 'office', ('2353445340', 1): 'admin'}
4
5  print(dogs)
6  print(tuples)
7
8  # We access directly to the value using its key
9  print(dogs['bc'])
10 print(telephones[23545344])
```

```
{'bc': 'border-collie', 'lr': 'labrador retriever', 'pg': 'pug'}
{('2353445340', 1): 'admin', ('23545344', 0): 'office'}
border-collie
John
```

In the next example, we can see that *dictionaries* are not a sorted sequence like *tuples* or *list*; and that the key can even be of various types within the same *dictionary*.

```
d = {1: 'first key', '2': 'second key', 23.0: 'third key',
     (23, 5): 'fourth key'}
print(d)
```

```
{1: 'first key', (23, 5): 'fourth key', '2': 'second key',
23.0: 'third key'}
```

Dictionaries are **mutable** data structures, *i.e*, its content can change through the execution of a program. There are two behaviors when we assign a value to key: 1) if the key does not exist, Python creates the key in the dictionary and assigns the value; and 2) if the key already exists, Python assigns the new value.

```
# If a key does not exist, Python creates a new one and assigns the value
dogs['te'] = 'terrier'
print(dogs)

# If the key already exist, Python just assigns the value
dogs['pg'] = 'pug-pug'
print(dogs)
```

```
{'bc': 'border-collie', 'lr': 'labrador retriever', 'te': 'terrier',
'pg': 'pug'}
{'bc': 'border-collie', 'lr': 'labrador retriever', 'te': 'terrier',
'pg': 'pug'}
```

We can also remove items directly from a *dictionary* by using the `del` sentence. For example:

```
# We remove items using del
del dogs['te']
print(dogs)
```

```
{'bc': 'border-collie', 'lr': 'labrador retriever', 'pg': 'pug-pug'}
```

We can check whether a given key exists in a *dictionary* by using the `in` statement. By default, every time we use this sentence directly with the name of the *dictionary*, Python assumes that we refer to its list of keys. The result of using `in` is `True` if the required key exists in the *dictionary* keys, and `False` otherwise:

```
# The sentence in checks whether exist a specific key in the dictionary
print('pg' in dogs)
print('te' in dogs)
```

```
True
False
```

Similarly, dictionaries have the `get()` method that allows us to check whether a key exist or not. This method requires two parameters: the key and a default value for missing keys. We can use any Python object as a default value.

```
# The method get() checks whether a key exists or not.

print(dogs.get('pg', False))  # logical value as default
print(dogs.get('te', 2))  # int value as default
print(dogs.get('te', 'The dog does not exist'))  # String value as default
```

```
True
False
pug-pug
```

This verification is useful when we need to build a *dictionary* during the execution of our program, such as we show in the following example:

```
# A string to be verified and a empty dictionary to count vowels
msg = 'supercalifragilisticexpialidocious'
vowels = dict()

for v in msg:
    if v not in 'aeiou':  # check wheter v is vowel or not
        continue

    # If v exist, add a key named as v initialized at 0
    if v not in vowels:
        vowels[v] = 0

    # If v alread exist, increase the counter
    vowels[v] += 1

```

```
16 print(vowels)
```

```
{'i': 7, 'a': 3, 'o': 2, 'e': 2, 'u': 2}
```

There are three useful methods that let us to retrieve the dictionary contents at different levels. These methods are: `keys()`, `values()`, and `items()`. The example below shows the results of using these methods and their outputs:

```
1 currency = {'Chile': 'Peso', 'Brazil': 'Real',
2             'Peru': 'Sol', 'Spain': 'Euro', 'Italy': 'Euro'}
3
4 print(currency.keys())  # returns a list with the keys
5 print(currency.values())  # returns a list with the values
6 print(currency.items())  # returns a list of tuples key-value
```

```
dict_keys(['Chile', 'Spain', 'Italy', 'Peru', 'Brazil'])
dict_values(['Peso', 'Euro', 'Euro', 'Sol', 'Real'])
dict_items([('Chile', 'Peso'), ('Spain', 'Euro'), ('Italy', 'Euro'),
            ('Peru', 'Sol'), ('Brazil', 'Real')])
```

The methods `keys()`, `values()`, and `items()` described above are completely suitable during iteration over dictionaries. By default, when we use a `for` loop to iterate over a *dictionary* Python iterates directly over the keys. In each iteration the local variable takes a key value in the list.

```
1 # Iteration over a dictionary
2
3 # By default Python iterates directly over the keys
4 print('This dictionary has the following keys:')
5
6 for k in currency:
7     print('{0}'.format(k))
```

```
This dictionary has the following keys:
Chile
Spain
Italy
Peru
Brazil
```

Or we can also use the method `keys()` explicitly

```
# Although we can also use the method keys()
print('This dictionary has the following keys:')

for k in currency.keys():
    print('{0}'.format(k))
```

```
This dictionary has the following keys:
Chile
Spain
Italy
Peru
Brazil
```

The method `values()` allows us to iterate over the list of values associated to each key.

```
# We use the method values() when we want to iterates using the values
print('The values in the dictionary:')
for v in currency.values():
    print('{0}'.format(v))
```

```
The values in the dictionary:
Peso
Euro
Euro
Sol
Real
```

Finally, the method `items()` provide us with a way to iterate using the pair key-value.

```
# The method items() allows us to retrieve a tuple (key, value)
print('The pairs key-value:')

for k, v in currency.items():
    print('the currency in {0} is {1}'.format(k, cv))
```

```
The pairs key-value:
the currency in Chile is Peso
the currency in Spain is Euro
```

```
the currency in Italy is Euro
the currency in Peru is Sol
the currency in Brazil is Real
```

Defaultdicts

Python provide us with a special case of *dictionary* called `defaultdicts` that allows us to assign a default value to the nonexistent keys. This type of *dictionary* is part of the `collections` library and let us to save time writing line codes to manage the cases when our code tries to access a nonexistent keys. The `defaultdics` accept also a function as default value, which can receive an action and return any object as key in the *dictionary*. For example, suppose we want to create a *dictionary* where each new key has as value a list with a string indicating the current number of items in the *dictionary*.

```
1  from collections import defaultdict
2
3  num_items = 0
4
5  def my_function():
6      global num_items
7      num_items += 1
8      return ([str(num_items)])
9
10 d = defaultdict(my_function)
11
12 print(d['a'])
13 print(d['b'])
14 print(d['c'])
15 print(d['d'])
16 print(d)
```

```
['1']
['2']
['3']
['4']
defaultdict(
  <function my_function at 0x000000000295CBF8>,
```

```
{'d': ['4'], 'b': ['2'], 'c': ['3'], 'a': ['1']})
```

Sets

A *set* is a data structure that contains an unordered collection of unique and immutable objects. For example: imagine that we have a list that contains a collection of tuples of items `(song, artist)` where different songs may be associated with the same artist, and we would like to build a list of all unique artists in our library. To do so, we may create a new list and for each item in the collection check if we already added the artist to the new list. That iteration is inefficient. *Sets* provide us with a way to do this task easily because the data structures make sure that each item in the *set* is unique even if we add the same item again.

Python requires that the *sets* contain **hashable** objects, *i.e*, an immutable object that has a hash value registered in the `__hash__()` method. This values never changes during its lifetime and can be compared to other objects. Hashable objects have the advantage that they can be used as keys in dictionaries.

```
songs_list = [("Uptown Funk", "Mark Ronson"),
              ("Thinking Out Loud", "Ed Sheeran"),
              ("Sugar", "Maroon 5"),
              ("Patterns In The Ivy", "Opeth"),
              ("Take Me To Church", "Hozier"),
              ("Style", "Taylor Swift"),
              ("Love Me Like You Do", "Ellie Goulding")]

artists = set()

for song, artist in songs_list:
    # The add() method append a new item to the set. We do not require to
    # check whether the item exist or not previously in the set.
    artists.add(artist)

print(artists)
```

```
{'Mark Ronson', 'Opeth', 'Hozier', 'Ed Sheeran', 'Maroon 5', 'Taylor Swift',
  'Ellie Goulding'}
```

We also can build a *set* using brackets, where a coma must separate the items. An empty *set* has to be created with the `set()` statement, otherwise a dictionary will be created.

```
# We can create a set using brackets including items separated by coma.
songs = {'Style', 'Uptown Funk', 'Take Me To Church', 'Sugar',
         'Thinking Out Loud', 'Patterns In The Ivy', 'Love Me Like You Do'}

print(songs)
print('Sugar' in songs)

for artist in artists:
    print("{} plays excellent music".format(artist))
```

```
{'Patterns In The Ivy', 'Take Me To Church', 'Sugar', 'Love Me Like You Do',
 'Style', 'Uptown Funk', 'Thinking Out Loud'}
True
Mark Ronson plays excellent music
Opeth plays excellent music
Hozier plays excellent music
Ed Sheeran plays excellent music
Maroon 5 plays excellent music
Taylor Swift plays excellent music
Ellie Goulding plays excellent music
```

Note that the results of the previous example show that the items in the *set* are unordered (similar to dictionaries). *Sets* cannot be indexed to retrieve their items. We can build an ordered list from a *set* as follows:

```
# Build a list from a set
artists_list = list(artists)
artists_list.sort()
print(artists_list)
```

```
['Ed Sheeran', 'Ellie Goulding', 'Hozier', 'Mark Ronson', 'Maroon 5',
'Opeth', 'Taylor Swift']
```

Sets data structures behave as mathematical *sets* and provide us with the same mathematical operations.

```
# Mathematical Operations
my_artists = {
    'Hozier', 'Opeth', 'Ellie Goulding', 'Mark Ronson', 'Taylor Swift'
}
```

```

artists_album = {'Maroon 5', 'Taylor Swift', 'Amy Wadge'}

print("All: {}".format(my_artists.union(artists_album)))
print("both: {}".format(artists_album.intersection(my_artists)))
```

```
All: {'Mark Ronson', 'Opeth', 'Hozier', 'Taylor Swift', 'Maroon 5',
'Amy Wadge', 'Ellie Goulding'}
both: {'Taylor Swift'}
```

```
# The A.difference(B) returns a set of items that exist only in A but
# not in B.
print("Only in A: {}".format(my_artists.difference(artists_album)))
```

```
Only in A: {'Ellie Goulding', 'Mark Ronson', 'Hozier', 'Opeth'}
```

```
# The symmetric difference returns a set of items that exist only in
# one of the sets, but not both.
print("Any but not both: {}".format(my_artists.symmetric_difference(
    artists_album)))
```

```
Any but not both: {'Mark Ronson', 'Opeth', 'Hozier', 'Maroon 5',
'Amy Wadge', 'Ellie Goulding'}
```

```
# Operation Order
bands = {"Opeth", "Guns N' Roses"}

print("my_artist is to bands:")
print("issuperset: {}".format(my_artists.issuperset(bands)))
print("issubset: {}".format(my_artists.issubset(bands)))
print("difference: {}".format(my_artists.difference(bands)))

print("-" * 20)

print("bands is to my_artists:")
print("issuperset: {}".format(bands.issuperset(my_artists)))
print("issubset: {}".format(bands.issubset(my_artists)))
print("difference: {}".format(bands.difference(my_artists)))
```

```
my_artist is to bands:
issuperset: False
issubset: False
difference: {'Mark Ronson', 'Taylor Swift', 'Hozier', 'Ellie Goulding'}
--------------------
bands is to my_artists:
issuperset: False
issubset: False
difference: {"Guns N' Roses"}
```

In some methods, the arguments' order does not matter, for example, `my_artists.union(artists_album)` returns the same result that `my_artist_album.union(my_artist)`. There are other methods where the arguments' order does matter, for example, `issubset()` and `issuperset()`.

2.2 Node-based Data Structures

In this section, we describe a set of data structures based on a single and basic structure called **node**. A **node** allocates an item and its elements and maintains one or more reference to neighboring nodes to represent more complex data structures collectively. One relevant aspect of these complex structure is the way on how we walk through each node. The **traversal** is the way to visit all the nodes in a node-base structure systematically. The following sections show how to build and traverse two essentials node-based structures: *linked lists* and, *trees*.

Singly Linked List

This data structure is one of the primary node-based structure. In a *linked list*, a collection of nodes forms a linear sequence where each node has a unique precedent and subsequent nodes. The first node is called `head` and the last node is called `tail`. In this structure, nodes have references to their `value` and to the `next` element in the sequence. Figure 2.11 shows a diagram of a linked list. In the `tail` node there is no reference to the `next` object.

The way to traverse a linked list is node-by-node recursively. Every time we get a node we have to pick the next one, indicated with the `next` statement. The traverse stops when there are no more nodes in the sequence. The code below shows how to build a linked list. Lines 21 to 31 show how to traverse the structure.

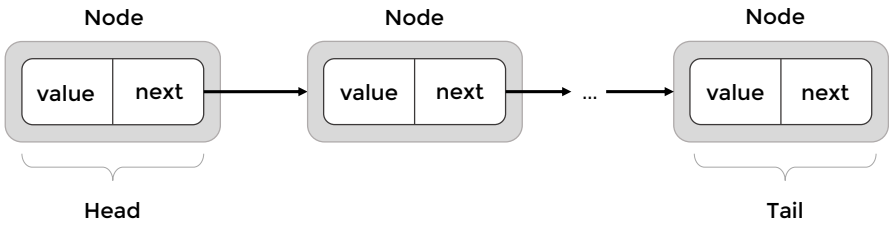

Figure 2.11: The simpler implementation of a linked list consists into a node that has two attributes: the value of the node and a reference to the next node. We can put as many nodes as we require.

```
class Node:
    # This class models the basic structure, the node.
    def __init__(self, value=None):
        self.next = None
        self.value = value

class LinkedList:
    # This class implement a singly linked list
    def __init__(self):
        self.tail = None
        self.head = None

    def add_node(self, value):
        if not self.head:
            self.head = Node(value)
            self.tail = self.head
        else:
            self.tail.next = Node(value)
            self.tail = self.tail.next

    def __repr__(self):
        rep = ''
        current_node = self.head

        while current_node:
            rep += '{0}'.format(current_node.value)
            current_node = current_node.next
            if current_node:
                rep += ' -> '

        return rep

if __name__ == '__main__':
    l = LinkedList()
    l.add_node(2)
```

```
36      l.add_node(4)
37      l.add_node(7)
38
39      print(l)
```

```
2 -> 4 -> 7
```

Trees

Trees are one of the most important data structure in computer science. A **tree** is a collection of nodes structured **hierarchically**. Opposite to the array-based structures (*e.g.* stacks and queues), the nodes that represent the items lay ordered **above** and **below** according to the **parent-child** hierarchy. A tree has a top node called **root** that is the only node that does not have a parent. Nodes other than the **root** have a single parent and one o more children. Children nodes descending from the same parent are called **siblings**.

We say that a node a is an **ancestor** of node b if a is in the path from b to the root. Nodes that have no children are called **leaf** nodes (sometimes called **external**). Nodes that are not the root or leaves are called **internal** nodes. Recursively, every node can be the root of its subtree. Figure 2.12 shows a tree representation of the animal kingdom. The root node has two children: Vertebrates and Invertebrates. The Invertebrates node has three children that are siblings each other: mollusks, arthropod, and worms. The node *Annelids* is a leave node. *Vertebrates* can be a root node of the subtree formed by its children.

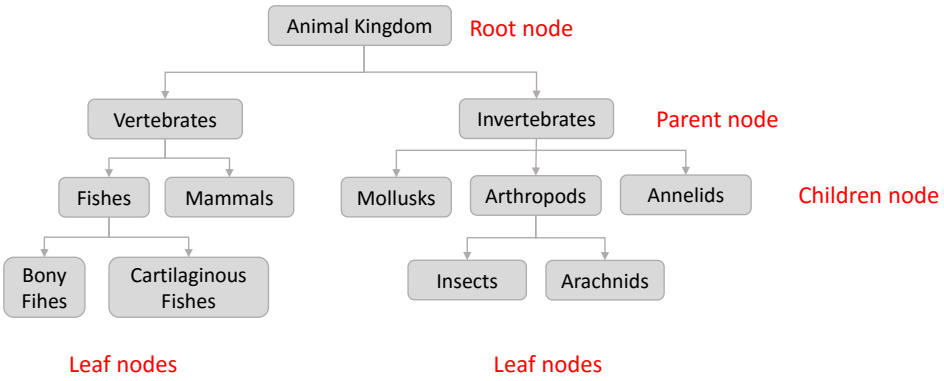

Figure 2.12: An example of a tree structure to represent the Animal Kingdom shows a tree representation of the animal kingdom. According to the definition, the root node has no a parent node. All the internal nodes have a parents-child relationship. The leaf nodes have no children. Every node can be the root of its own subtree, *e.g*, Fishes.

An **edge** connects a pair of nodes *(u, v)* that have a parent-child relationship. Each node has a unique incoming edge (parent) and zero or various outgoing edges (children). An ordered sequence of consecutive nodes joint by a set of

edges from a starting node to a destination node through the tree form a **path**. In the same Figure 2.12, there are two edges that connect the node *Fishes* with its children *Bony* and *Cartilaginous*. The set of edges from *Bony* to *Animals* form the path *Animals-Vertebrates-Fishes-Bony*.

The **depth** of a node *b* is the number of levels or ancestors that exist between *b* and the root node. The **height** of a tree is the number of levels in the tree, or the maximum depth reached among the leaf nodes. As shown in Figure 2.12, the *Fishes* node has depth 2, and the height of the tree is 3.

Binary Tree

Binary trees are among the most used tree structures in computer science. In a binary tree, each node has a maximum number of two children; each child node has a label: **left- child** and **right-child**, and regarding precedence, the left child precedes the right child.

In binary trees, the number of nodes grows exponentially with depth. Let d be the **level** of an binary tree **T** defined as the set of nodes located at the same depth d. At the level $d = 0$ there is at least only one node (the root). The level $d = 1$ has at least two nodes, and so on. At any level d, the tree has a maximum number of 2^d levels. The special case when every node has two children is know as a **complete tree**.

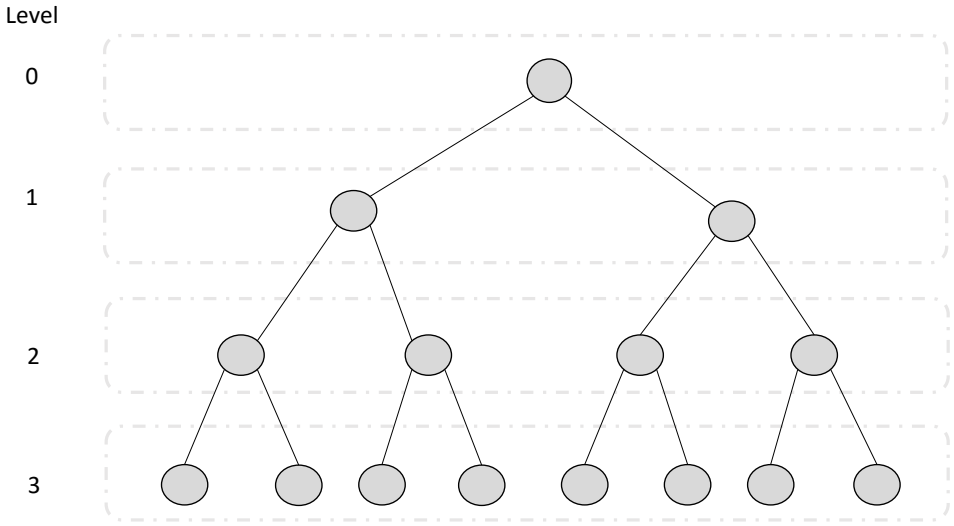

Figure 2.13: An example of a binary tree. As we can see, the node 0 is the root of the tree. For this example, we adopt the convention of setting the numbers while traversing the tree in amplitude.

A practical example of binary trees is **decision trees**. In this kind of trees, each interior node and the root represent a query, and their outgoing edges represent the possible answers. Another example is **expression trees**; they represent arithmetic operations where variables correspond to leaf nodes and operators to interior nodes.

Linked Structure Based Binary Tree

The linked structure based binary tree correspond to the recursive version for binary trees. Each node of the tree is an object where each attribute is a reference to the parent node, children nodes, and its value. We use `None` to indicate that an attribute does not exist. For example, if we write the root node, the attribute `parent` is equal to `None`. Now we show the implementation of a binary tree using a linked structure:

```
class Node:

    def __init__(self, value, parent=None):
        self.value = value
        self.parent = parent
        self.left_child = None
        self.right_child = None

    def __repr__(self):
        return 'parent: {0}, value: {1}'.format(self.parent, self.value)


class BinaryTree:

    def __init__(self, root_node=None):
        self.root_node = root_node

    def add_node(self, value):
        if self.root_node is None:
            self.root_node = Node(value)
        else:
            temp = self.root_node
            added = False

            while not added:
                if value <= temp.value:
                    if temp.left_child is None:
                        temp.left_child = Node(value, temp.value)
                        added = True
```

```
30
31                      else:
32                          temp = temp.left_child
33
34                  else:
35                      if temp.right_child is None:
36                          temp.right_child = Node(value, temp.value)
37                          added = True
38
39                      else:
40                          temp = temp.right_child
41
42      def __repr__(self):
43          def traverse_tree(node, side="root"):
44              ret = ''
45
46              if Node is not None:
47                  ret += '{0} -> {1}\n'.format(node, side)
48                  ret += traverse_tree(node.left_child, 'left')
49                  ret += traverse_tree(node.right_child, 'right')
50
51              return ret
52
53          return traverse_tree(self.root_node)
54
55
56  T = BinaryTree()
57  T.add_node(4)
58  T.add_node(1)
59  T.add_node(5)
60  T.add_node(3)
61  T.add_node(20)
62
63  print(T)
```

```
parent: None, value: 4 -> root
```

```
parent: 4, value: 1 -> left
parent: 1, value: 3 -> right
parent: 4, value: 5 -> right
parent: 5, value: 20 -> right
```

Binary Tree Traversal

In the following sections, we describe the three basic methods to traverse a binary tree: pre-order traversal, in-order traversal, post-order traversal.

Pre-Order Traversal

In this method we first visit the root node and then its children recursively:

```
# 31_binary_trees_pre_order_traversal.py

class BinaryTreePreOrder(BinaryTree):
    # We inherited the original class of our binary tree, and override the
    # __repr__ method to traverse the tree using pre-order traversal.

    def __repr__(self):
        def traverse_tree(node, side="root"):
            ret = ''

            if node is not None:
                ret += '{0} -> {1}\n'.format(node, side)
                ret += traverse_tree(node.left_child, 'left')
                ret += traverse_tree(node.right_child, 'right')

            return ret

        return traverse_tree(self.root_node)


if __name__ == '__main__':
    # We add some nodes to the tree
    T = BinaryTreePreOrder()
```

```
24      T.add_node(4)
25      T.add_node(1)
26      T.add_node(5)
27      T.add_node(3)
28      T.add_node(20)
29
30      print(T)
```

```
parent: None, value: 4 -> root
parent: 4, value: 1 -> left
parent: 1, value: 3 -> right
parent: 4, value: 5 -> right
parent: 5, value: 20 -> right
```

In-Order Traversal

In this method we first visit the **left-child**, then the **root** and finally the **right-child** recursively:

```
1  # 32_binary_trees_in_order_traversal.py
2
3  class BinaryTreeInOrder(BinaryTree):
4      # We inherited the original class of our binary tree, and override the
5      # __repr__ method to traverse the tree using pre-order traversal.
6
7      def __repr__(self):
8          def traverse_tree(node, side="root"):
9              ret = ''
10
11             if node is not None:
12                 ret += traverse_tree(node.left_child, 'left')
13                 ret += '{0} -> {1}\n'.format(node, side)
14                 ret += traverse_tree(node.right_child, 'right')
15
16             return ret
17
18         return traverse_tree(self.root_node)
19
```

```
20
21 if __name__ == '__main__':
22     # We add some nodes to the tree
23     T = BinaryTreeInOrder()
24     T.add_node(4)
25     T.add_node(1)
26     T.add_node(5)
27     T.add_node(3)
28     T.add_node(20)
29
30     print(T)
```

```
parent: 4, value: 1 -> left
parent: 1, value: 3 -> right
parent: None, value: 4 -> root
parent: 4, value: 5 -> right
parent: 5, value: 20 -> right
```

Post-Order Traversal

The post-order traversal first finds the sub-trees descending from the children nodes, and finally the root.

```
1  # 33_binary_trees_post_order_traversal.py
2
3  class BinaryTreePostOrder(BinaryTree):
4      # We inherited the original class of our binary tree, and override the
5      # __repr__ method to traverse the tree using pre-order traversal.
6
7      def __repr__(self):
8          def traverse_tree(node, side="root"):
9              ret = ''
10
11             if node is not None:
12                 ret += traverse_tree(node.left_child, 'left')
13                 ret += traverse_tree(node.right_child, 'right')
14                 ret += '{0} -> {1}\n'.format(node, side)
15
```

```
            return ret

        return traverse_tree(self.root_node)


if __name__ == '__main__':
    # We add some nodes to the tree
    T = BinaryTreePostOrder()
    T.add_node(4)
    T.add_node(1)
    T.add_node(5)
    T.add_node(3)
    T.add_node(20)

    print(T)
```

```
parent: 1, value: 3 -> right
parent: 4, value: 1 -> left
parent: 5, value: 20 -> right
parent: 4, value: 5 -> right
parent: None, value: 4 -> root
```

N-ary Trees

The N-ary trees correspond to a generalization of trees. Differently from the binary case, in N-ary trees, each node may have zero or more children.

Linked Structured N-ary Tree

Similar to a binary tree, we can build N-ary trees using a linked structure where each node is a tree itself. Following the tree definition, the complete N-ary tree is a collection of nodes that we append incrementally. Each node includes the following attributes: `node_id`, `parent_id`, `children`, and `value`. Figure 2.14 shows an example of a tree with three levels where each node has a value and an identifier.

The code below shows a recursive implementation of a linked structured tree:

```
# 34_linked_trees.py
```

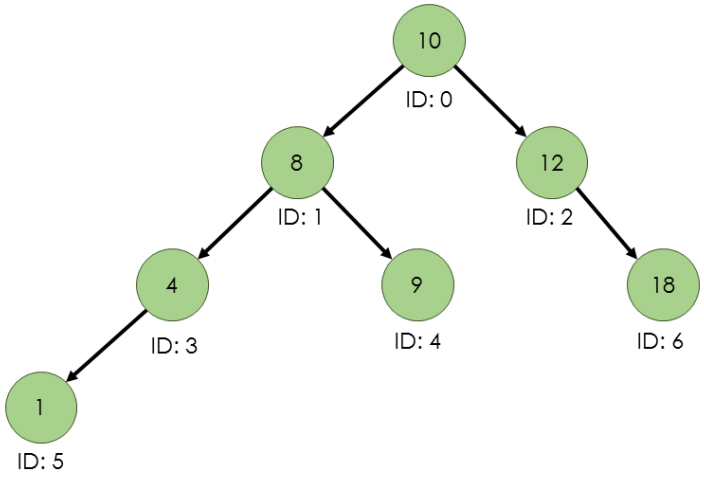

Figure 2.14: An example of a general tree structure. Green circles denote the nodes that include its value. Each node also has an identification number ID. Black arrows represent edges.

```
2
3  class Tree:
4      # We create the basic structure of the tree. Children nodes can be keep in
5      # a different data structure, such as: a lists or a dictionary. In this
6      # example we manage the children nodes in a dictionary.
7
8      def __init__(self, node_id, value=None, parent_id=None):
9          self.node_id = node_id
10         self.parent_id = parent_id
11         self.value = value
12         self.children = {}
13
14     def add_node(self, node_id, value=None, parent_id=None):
15         # Every time that we add a node, we need to verify the parent of the
16         # new node. If it is not the parent, we search recursively through the
17         # the tree until we find the right parent node.
18
19         if self.node_id == parent_id:
20             # If the node is the parent, we update the dictionary with the
21             # children.
22             self.children.update({node_id: Tree(node_id, value, parent_id)})
```

```
23          else:
24              # If the node is not the parent we search recursively
25              for child in self.children.values():
26                  child.add_node(node_id, value, parent_id)
27
28      def get_node(self, node_id):
29          # We recursively get the node as long as it exists in the tree.
30          if self.node_id == node_id:
31              return self
32          else:
33              for child in self.children.values():
34                  node = child.get_node(node_id)
35                  if node:
36                      # if the node exists in the tree, returns the node
37                      return node
38
39      def __repr__(self):
40          # We override this method in order to traverse recursively the node in
41          # the tree.
42          def traverse_tree(root):
43              ret = ''
44              for child in root.children.values():
45                  ret += "id-node: {} -> parent_id: {} -> value: {}\n".format(
46                      child.node_id, child.parent_id, child.value)
47                  ret += traverse_tree(child)
48              return ret
49
50          ret = 'root:\nroot-id: {} -> value: {}\n\nchildren:\n'.format(
51              self.node_id, self.value)
52          ret += traverse_tree(self)
53
54          return ret
55
56
57  if __name__ == '__main__':
```

```
58      T = Tree(0, 10)
59      T.add_node(1, 8, 0)
60      T.add_node(2, 12, 0)
61      T.add_node(3, 4, 1)
62      T.add_node(4, 9, 1)
63      T.add_node(5, 1, 3)
64      T.add_node(6, 18, 2)
```

```
root:
root-id: 0 -> value: 10

children:
id-node: 1 -> parent_id: 0 -> value: 8
id-node: 3 -> parent_id: 1 -> value: 4
id-node: 5 -> parent_id: 3 -> value: 1
id-node: 4 -> parent_id: 1 -> value: 9
id-node: 2 -> parent_id: 0 -> value: 12
id-node: 6 -> parent_id: 2 -> value: 18
```

We use the method `get_node()` to get a specific node. In this implementation, the method returns the object representing the node:

```
1       node = T.get_node(6)
2       print('The ID of the node is {}'.format(node))
3
4       node = T.get_node(1)
5       print('The node has {} children'.format(len(node.children)))
```

```
The ID of the node is root:
root-id: 6 -> value: 18

children:

The node have 2 children
```

N-ary Tree Traversal

There are two basic methods to traverse the tree: pre-order traversal and post-order traversal. These approaches generalize the traverse methods for binary trees. Note that in this case, the in-order traverse is difficult to define because we cannot determine after which child we have to visit the root node.

Pre-Order Traversal

In this method, we first visit the root node and then its children recursively:

```
# 35_trees_pre_order_traversal.py

class TreePreOrder(Tree):
    # We inherited the original class of our linked tree, and override the
    # __repr__ method to traverse the tree using pre-order traversal.

    def __repr__(self):
        def traverse_tree(root):
            ret = ''
            # We first visit the root note
            ret += "node_id: {}, parent_id: {} -> value: {}\n".format(
                root.node_id, root.parent_id, root.value)

            # And finally, we traverse the children recursively
            for child in root.children.values():
                ret += traverse_tree(child)

            return ret

        return traverse_tree(self)


if __name__ == '__main__':
    # We add some nodes to the tree
    T = TreePreOrder(0, 10)
    T.add_node(1, 8, 0)
    T.add_node(2, 12, 0)
```

```
28      T.add_node(3, 4, 1)
29      T.add_node(4, 4, 1)
30      T.add_node(5, 1, 3)
31      T.add_node(6, 18, 2)
32
33      print(T)
```

```
node_id: 0, parent_id: None -> value: 10
node_id: 1, parent_id: 0 -> value: 8
node_id: 3, parent_id: 1 -> value: 4
node_id: 5, parent_id: 3 -> value: 1
node_id: 4, parent_id: 1 -> value: 4
node_id: 2, parent_id: 0 -> value: 12
node_id: 6, parent_id: 2 -> value: 18
```

Post-Order Traversal

The post-order traversal first finds the sub-trees descendant from the children nodes, and finally the root:

```
1  class TreePostOrder(Tree):
2      # We inherited the class Tree from te previous example and we override the
3      # __repr__ method using the post-order traversal.
4
5      def __repr__(self):
6          def traverse_tree(root):
7              ret = ''
8
9              # we first recursively traverse the children
10             for child in root.children.values():
11                 ret += traverse_tree(child)
12
13             # Finally, we visit the root note
14             string = "node_id: {}, parent_id: {} -> value: {}\n"
15             ret += string.format(root.node_id, root.parent_id, root.value)
16
17             return ret
18
```

```
19          return traverse_tree(self)
20
21
22  if __name__ == '__main__':
23      # We add instances to the tree
24      T = TreePostOrder(0, 10)
25      T.add_node(1, 8, 0)
26      T.add_node(2, 12, 0)
27      T.add_node(3, 4, 1)
28      T.add_node(4, 4, 1)
29      T.add_node(5, 1, 3)
30      T.add_node(6, 18, 2)
31
32      print(T)
```

```
node_id: 5, parent_id: 3 -> value: 1
node_id: 3, parent_id: 1 -> value: 4
node_id: 4, parent_id: 1 -> value: 4
node_id: 1, parent_id: 0 -> value: 8
node_id: 6, parent_id: 2 -> value: 18
node_id: 2, parent_id: 0 -> value: 12
node_id: 0, parent_id: None -> value: 10
```

Non-Recursive Traversal

There are two non-recursive methods to implement the tree traversal: **Breadth-First Search** (**BFS**) and **Depth-First Search** (**DFS**). These non-recursive algorithms use auxiliary data structures to keep a record of the nodes we must visit, to avoid infinite loops while traversing trees or other complex node-based structures.

Breadth-First Search

In the **BFS** strategy, the algorithm traverses the nodes level by level hierarchically, *i.e*, the root node first, then the set of nodes on the second level, etc. This algorithm uses a queue to keep the nodes that it has to visit in the following iterations.

```
1
2  class TreeBFS(Tree):
```

```
3       # We inherited the original class Tree from the previous example and
4       # override the __repr__ method to apply the BFS algorithm.
5
6       def __repr__(self):
7           def traverse_tree(root):
8               ret = ''
9               Q = deque()
10              Q.append(root)
11
12              # We use a list to keep the visited nodes
13              visited = []
14
15              while len(Q) > 0:
16                  p = Q.popleft()
17
18                  if p.node_id not in visited:
19                      # We check if the node is in the visited nodes list. If is
20                      # not in the list, we add it
21                      visited.append(p.node_id)
22
23                      ret += "node_id: {}, parent_id: {} -> value: {}\n".format(
24                          p.node_id, p.parent_id, p.value)
25                      for child in p.children.values():
26                          Q.append(child)
27
28              return ret
29          return  traverse_tree(self)
30
31
32  if __name__ == '__main__':
33      # We add items to the tree
34      T = TreeBFS(0, 10)
35      T.add_node(1, 8, 0)
36      T.add_node(2, 12, 0)
37      T.add_node(3, 4, 1)
```

```
    T.add_node(4, 4, 1)
    T.add_node(5, 1, 3)
    T.add_node(6, 18, 2)

    print(T)
```

```
node_id: 0, parent_id: None -> value: 10
node_id: 1, parent_id: 0 -> value: 8
node_id: 2, parent_id: 0 -> value: 12
node_id: 3, parent_id: 1 -> value: 4
node_id: 4, parent_id: 1 -> value: 4
node_id: 6, parent_id: 2 -> value: 18
node_id: 5, parent_id: 3 -> value: 1
```

Depth-First Search

In the **DFS** approach, the traversal algorithm starts from the top node and then goes down until it reaches a leaf. To achieve this kind of traversal, we have to use a stack to add the children nodes that we will visit in future iterations:

```
class TreeDFS(Tree):
    # We inherited the original class Tree from the previous example and
    # override the __repr__ method to apply the BFS algorithm.

    def __repr__(self):
        def traverse_tree(root):
            ret = ''
            Q = []
            Q.append(root)

            # We use a list to keep the visited nodes
            visited = []

            while len(Q) > 0:
                p = Q.pop()

                if p.node_id not in visited:
                    # We check if the node is in the visited nodes list. If is
```

```
19                      # not in the list, we add it
20                      visited.append(p.node_id)
21
22                      ret += "node_id: {}, parent_id: {} -> value: {}\n".format(
23                          p.node_id, p.parent_id, p.value)
24                      for child in p.children.values():
25                          Q.append(child)
26
27              return ret
28          return traverse_tree(self)
29
30
31  if __name__ == '__main__':
32      # We add items to the tree
33      T = TreeDFS(0, 10)
34      T.add_node(1, 8, 0)
35      T.add_node(2, 12, 0)
36      T.add_node(3, 4, 1)
37      T.add_node(4, 4, 1)
38      T.add_node(5, 1, 3)
39      T.add_node(6, 18, 2)
40
41      print(T)
```

```
node_id: 0, parent_id: None -> value: 10
node_id: 2, parent_id: 0 -> value: 12
node_id: 6, parent_id: 2 -> value: 18
node_id: 1, parent_id: 0 -> value: 8
node_id: 4, parent_id: 1 -> value: 4
node_id: 3, parent_id: 1 -> value: 4
node_id: 5, parent_id: 3 -> value: 1
```

In this chapter, we reviewed the most common data structures in Python, if the reader wants to go deeper into data structures and algorithms we recommend the books of Cormen [4] and Karumanchi [5].

2.3 Hands-On Activities

Activity 2.1

In a production line of bottles, you have to implement software that lets the user predict the *output* of his factory. A colleague has modeled the process, and he asks you to finish the code by adding all the functionalities. Your **task** is to complete only the methods that appear commented in the code, explained below.

```
1  from collections import deque
2  from package import Package
3  from bottle import Bottle
4
5
6  class Machine:
7
8      def process(self, incoming_production_line):
9          print("----------------------")
10         print("Machine {} started working.".format(
11             self.__class__.__name__))
12
13
14 class BottleModulator(Machine):
15
16     def __init__(self):
17         self.bottles_to_produce = 0
18
19     def process(self, incoming_production_line=None):
20         super().process(incoming_production_line)
21         # --------------
22         # Complete the method
23         # --------------
24         return None
25
26
27 class LowFAT32(Machine):
28
29     def __init__(self):
```

```
30          self.discarded_bottles = []
31
32      def discard_bottle(self, bottle):
33          self.discarded_bottles.append(bottle)
34
35      def print_discarded_bottles(self):
36          print("{} bottles were discarded".format(
37              len(self.discarded_bottles)))
38
39      def process(self, incoming_production_line):
40          # -----------------
41          # Complete the method
42          # -----------------
43          return None
44
45
46  class HashSoda9001(Machine):
47
48      def process(self, incoming_production_line):
49          super().process(incoming_production_line)
50          # ----------------
51          # Complete the method
52          # ----------------
53          return None
54
55
56  class PackageManager(Machine):
57
58      def process(self, incoming_production_line):
59          packages = deque()
60          for stack in incoming_production_line.values():
61              package = Package()
62              package.add_bottles(stack)
63              packages.append(package)
64          return packages
```

```
65
66
67  class Factory:
68
69      def __init__(self):
70          self.bottlemodulator = BottleModulator()
71          self.lowFAT32 = LowFAT32()
72          self.hashSoda9001 = HashSoda9001()
73          self.packageManager = PackageManager()
74
75      def producir(self, num_bottles):
76          self.bottlemodulator.bottles_to_produce = num_bottles
77          product = None
78          for machine in [self.bottlemodulator,
79                          self.lowFAT32,
80                          self.hashSoda9001,
81                          self.packageManager]:
82              product = machine.process(product)
83          return product
84
85
86  if __name__ == "__main__":
87
88      num_bottles = 423
89
90      factory = Factory()
91      output = factory.producir(num_bottles)
92      print("----------------------")
93      print("{} bottles produced {} packages".format(
94          num_bottles, len(output)))
95      for package in output:
96          package.see_content()
97      print("----------------------")
```

The class ***bottle*** has a parameter that denote its maximum capacity in liters (lts). By default the maximum capacity is 1 lt. and it is filled with the delicious soda *DCC-Cola*. The class ***factory*** has a method that receives the number of bottles

that will be produced. Each machine has the method ***process*** that receives bottles as ***incoming_production_line***.

1. **Bottlemodulator**: this machine creates the bottles. It has an attribute to set up the number of bottles to produce. The ***incoming_production_line*** is null. By default, it produces bottles of 1-liter capacity, however, after a particular number of bottles occurs the following variations:

   - Each five produced bottles, the next bottle (ex: 6, 11, 16, ...) will have three times of the standard capacity (1 lt.)
   - Each six produced bottles, the next bottle (ex: 7, 13, 19, ...) will have half of the last bottle capacity (1 lt.) plus four times of the antepenultimate bottle.

   At the end of this process, each bottle passes to a production line in the same order that they were created. The method that models this machine returns the production line.

2. **Low-FAT32**: this machine processes the production line provided by the **Bottlemodulator** and fills a dispatch line. It chooses the first bottle from the incoming production line. If there is no bottle in this new line, the machine adds the first one. If the machine already has added bottles:

   - It puts the bottle at the end of the dispatch line if the bottle has the same or greater capacity than the last bottle in the line.
   - It puts the bottle at the beginning of the dispatch line if the bottle has the same or less capacity that the first bottle in the line.
   - It discards the bottle in other cases.

   At the end of this process, it shows the number of discarded bottles. Finally, the model of the machine returns the dispatch line.

3. **HashSoda9001**: this machine classifies and stacks the incoming bottles according to their size (for **n** different capacities we will have **b** different stacks).

4. **PackageManager**: the function of this machine is to pack the stacks of bottles provided by the previous machine. It returns a list of packages. Your colleague has already programmed this machine.

You have to complete the BottleModulator, Low-FAT32, and HashSoda9001 machines, such that they return the expected output.

Activity 2.2

Most of the data structures saw in this chapter are linear, however, in some situations, we require to work using more dimensions, such as a graph to navigate in a labyrinth or a tree to make decisions or perform depth search. In this activity, we use a data structure that models a subway map, where each station can have until four adjacent subway stations: *Right*, *Left*, *Up*, and *Down*. Below we show an example of a map. Note that station one is connected to station two, but not the other way around. We can also note that it is possible to reach station eleven from station zero. However, it is not feasible to go from station zero to station nineteen.

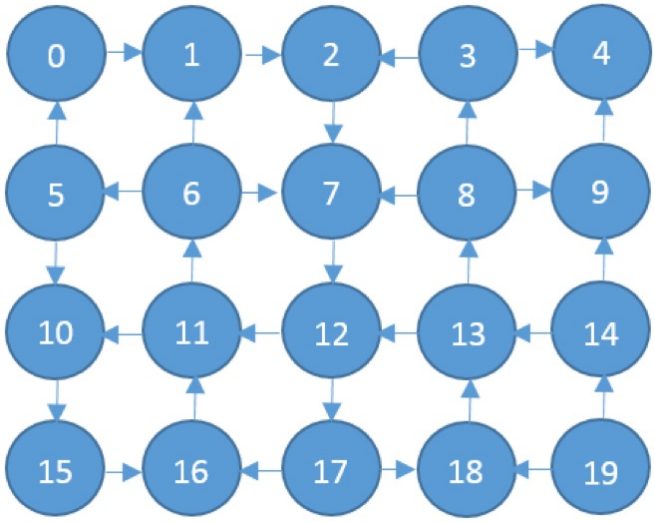

Figure 2.15

Get the files *main.py* and *station.py* provided in *https://github.com/advancedpythonprogramming*

Modify only the file *main.py* and complete the `path` method, that works as follows:

- This method receives two subway stations
- It returns **False** if it is not possible to arrive from the origin to the end. You must respect the rest of the paths.
- It returns **True** if exist a path between the origin and the destination. It also prints the path.

You can create methods, classes or anything you require in the *main.py* file. You cannot change the signature of the `path` method. You can work with the map provided in 2.15. The method used to generate the map is `SubwayMetro.example_map()`