Working with queues

DATA STRUCTURES AND ALGORITHMS IN PYTHON

Miriam Antona
Software Engineer

datacamp

Queues

- **FIFO:** First-In First-Out
  - **First inserted** item is the **first** to be **removed**

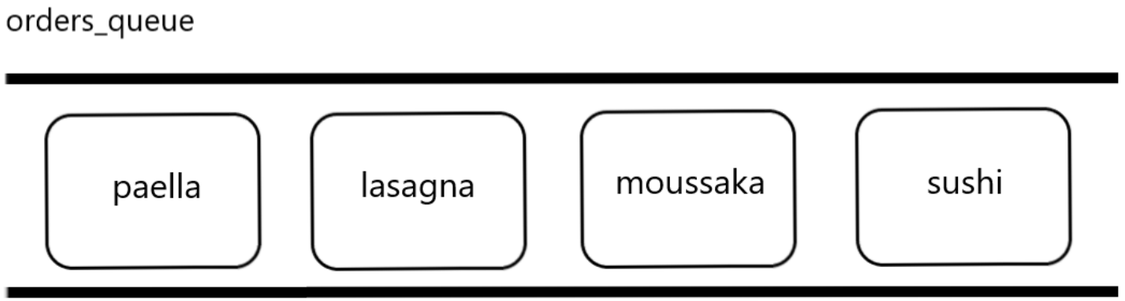
Queues

- **FIFO:** First-In First-Out
  - **First inserted** item is the **first** to be **removed**

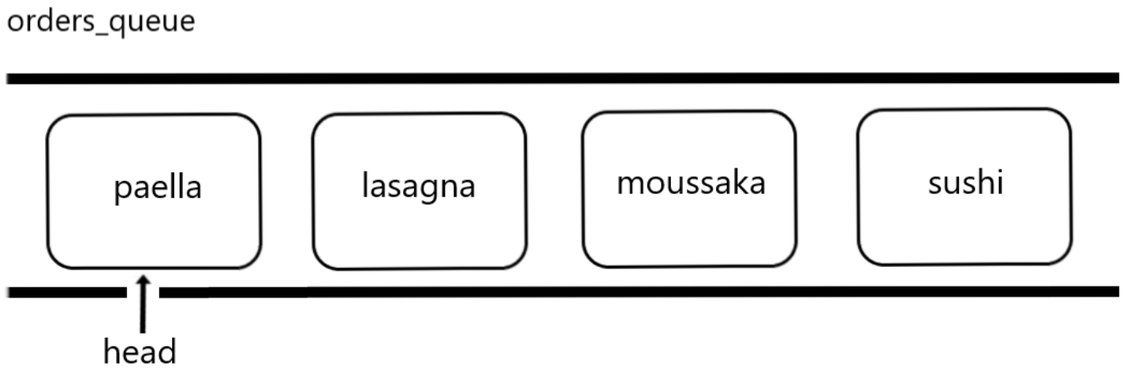
Queues

- **FIFO:** First-In First-Out
  - **First inserted** item is the **first** to be **removed**

Queues - structure

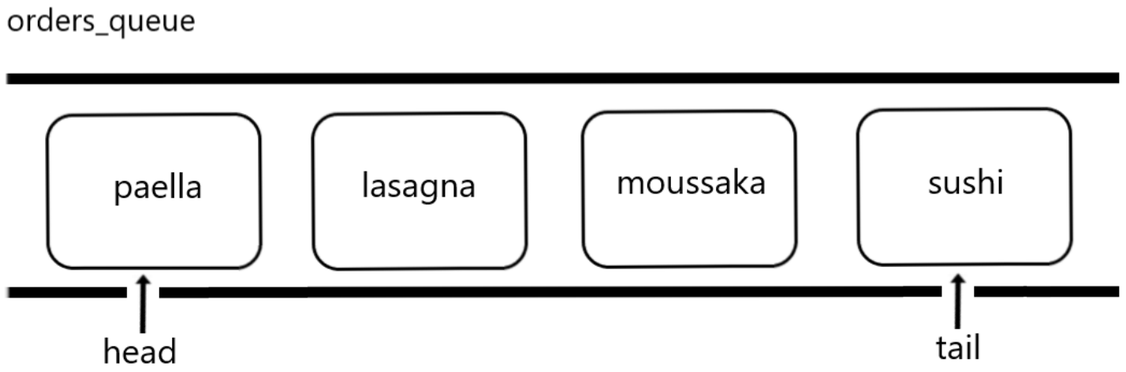


Queues - structure



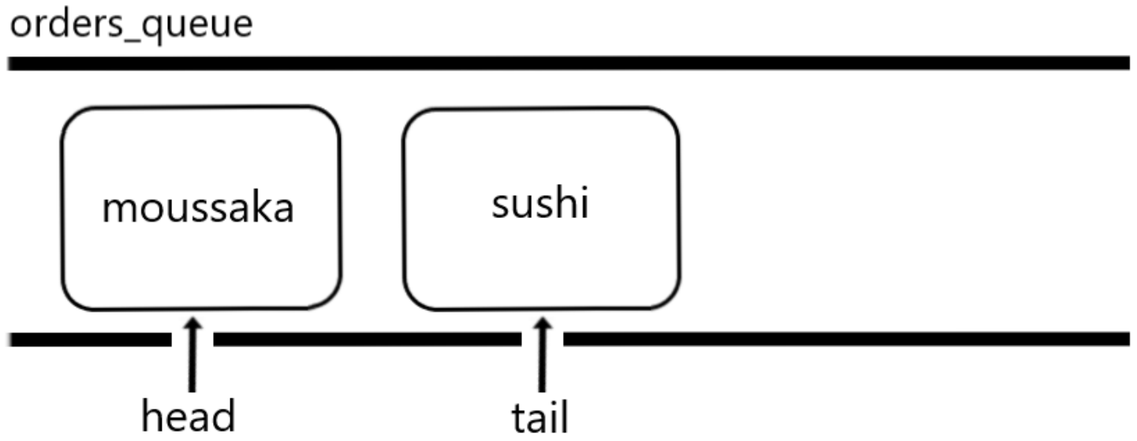
- Beginning: **head**

Queues - structure

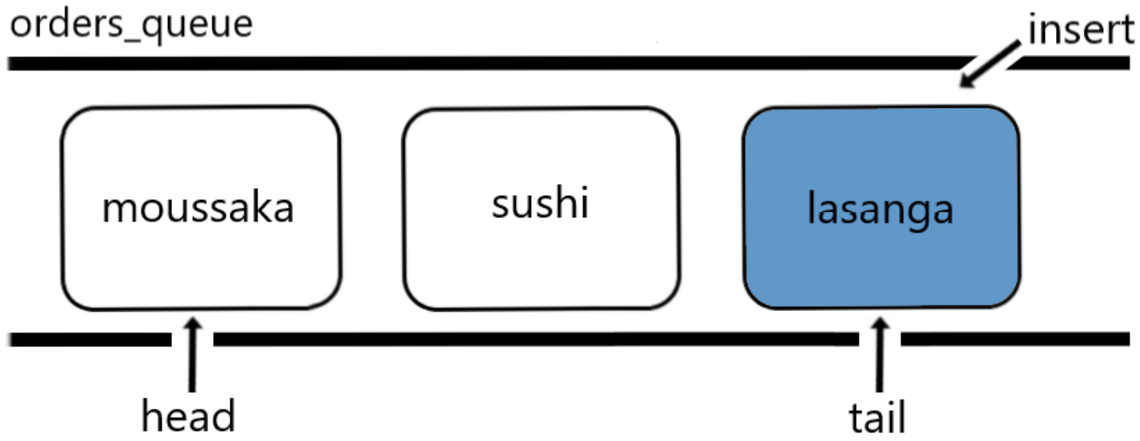


- Beginning: **head**
- End: **tail**

Queues - features

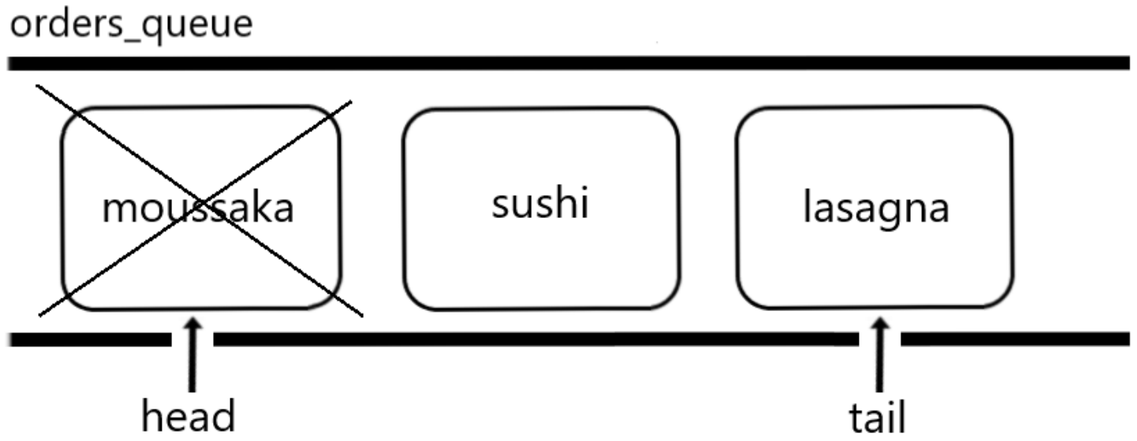


Queues - features



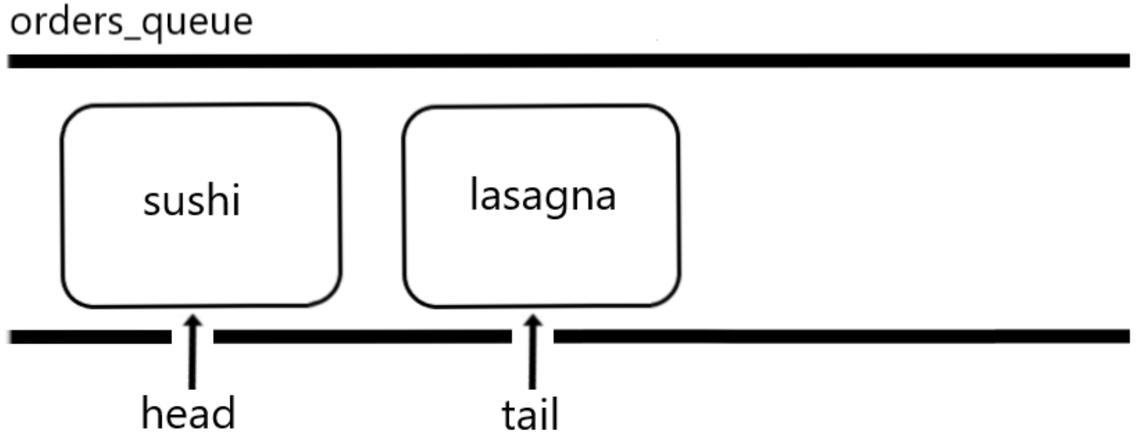
- Can only **insert** at the **end**
  - ***Enqueue***

Queues - features



- Can only **insert** at the **end**
  - ***Enqueue***
- Can only **remove** from the **head**

Queues - features



- Can only **insert** at the **end**
  - ***Enqueue***
- Can only **remove** from the **head**
  - ***Dequeue***
- Other kinds of queues:
  - Doubly ended queues
  - Circular queues
  - Priority queues

Queues - real-world use cases use

- **Printing tasks** in a printer
  - Documents are printed in the order they are received
- Applications where the **order of requests matters**
  - Tickets for a concert
  - Taxi services

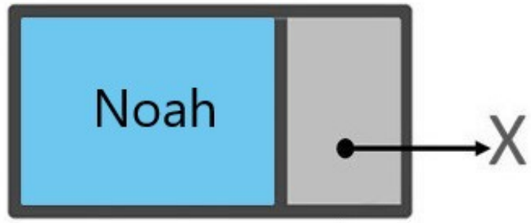
Queues - implementation

```
class Node:
  def __init__(self,data):
    self.data = data
    self.next = None
```

```
class Queue:
  def __init__(self):
    self.head = None
    self.tail = None
```

Queues - enqueue

```
def enqueue(self,data):
  new_node = Node(data)
  if self.head == None:
```

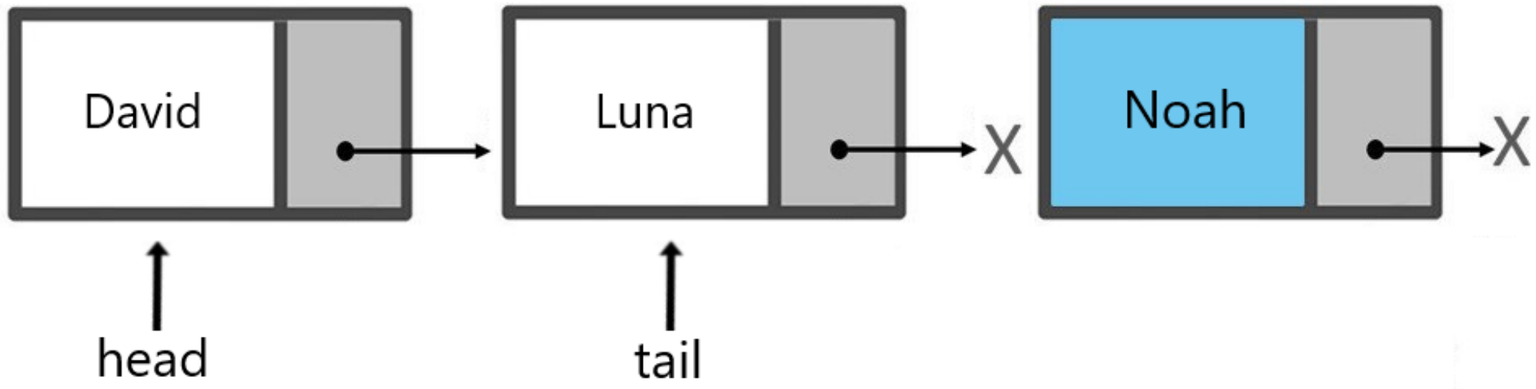


Queues - enqueue

```
def enqueue(self,data):
  new_node = Node(data)
  if self.head == None:
    self.head = new_node
    self.tail = new_node
```

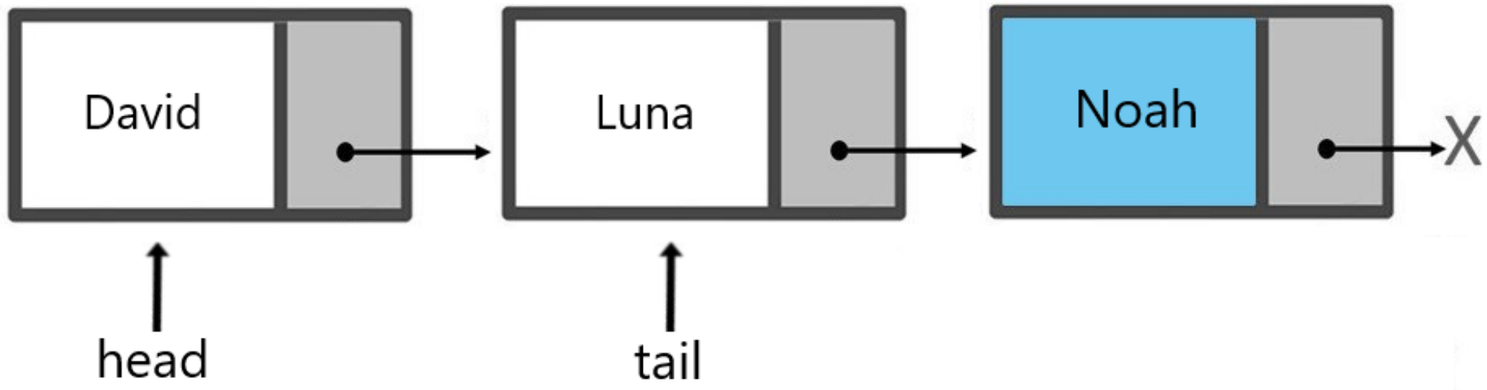
Queues - enqueue

```
def enqueue(self,data):
  new_node = Node(data)
  if self.head == None:
    self.head = new_node
    self.tail = new_node
  else:
```



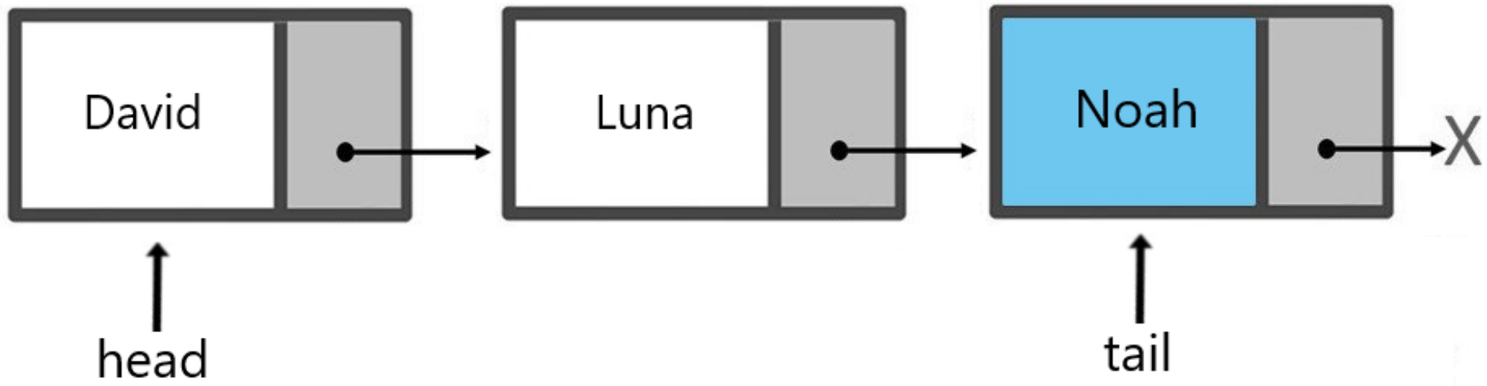
Queues - enqueue

```
def enqueue(self,data):
  new_node = Node(data)
  if self.head == None:
    self.head = new_node
    self.tail = new_node
  else:
    self.tail.next = new_node
```

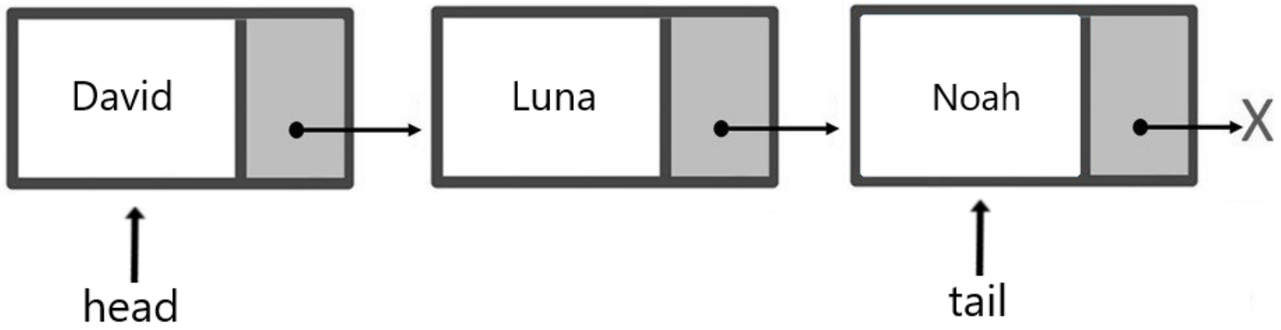


Queues - enqueue

```
def enqueue(self,data):
  new_node = Node(data)
  if self.head == None:
    self.head = new_node
    self.tail = new_node
  else:
    self.tail.next = new_node
    self.tail = new_node
```

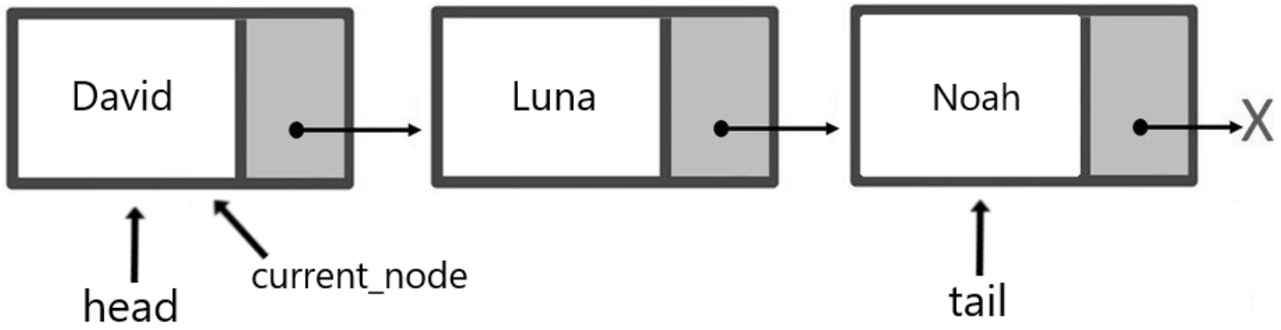


Queues - dequeue



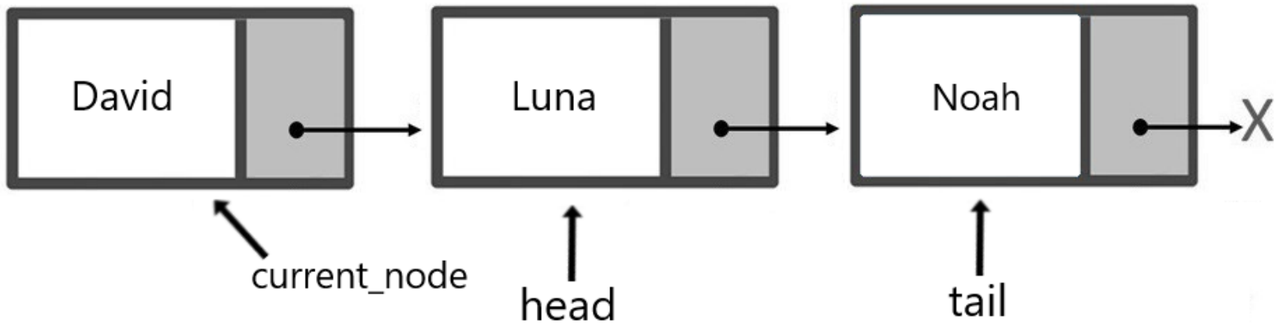

```
def dequeue(self):
  if self.head:
```

Queues - dequeue



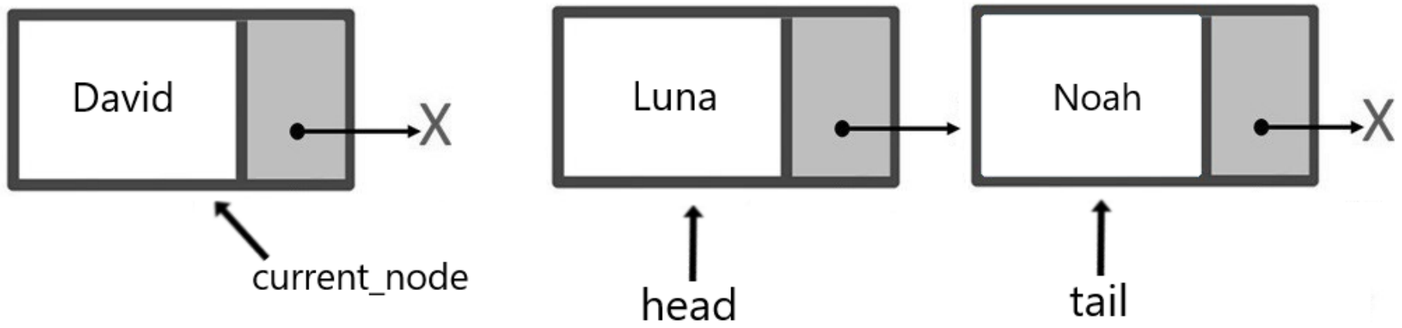

```
def dequeue(self):
  if self.head:
    current_node = self.head
```

Queues - dequeue



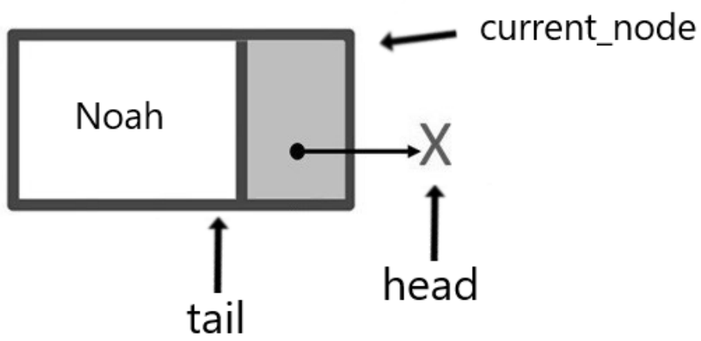

```
def dequeue(self):
  if self.head:
    current_node = self.head
    self.head = current_node.next
```

Queues - dequeue




```
def dequeue(self):
  if self.head:
    current_node = self.head
    self.head = current_node.next
    current_node.next = None
```

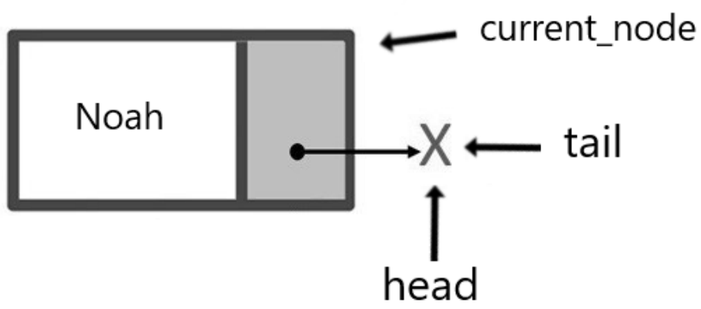
Queues - dequeue




```
def dequeue(self):
  if self.head:
    current_node = self.head
    self.head = current_node.next
    current_node.next = None

    if self.head == None:
```

Queues - dequeue




```
def dequeue(self):
  if self.head:
    current_node = self.head
    self.head = current_node.next
    current_node.next = None

    if self.head == None:
      self.tail = None
```

SimpleQueue in Python

- Module: **queue**
  - **Queue**
  - **SimpleQueue**

```
import queue

orders_queue = queue.SimpleQueue()

orders_queue.put("Sushi")
orders_queue.put("Lasagna")
orders_queue.put("Paella")

print("The size is: ", orders_queue.qsize())
```

```
The size is: 3
```

```
print(orders_queue.get())
print(orders_queue.get())
print(orders_queue.get())
```

```
Sushi
Lasagna
Paella
```

```
print("Empty queue: ", orders_queue.empty())
```

```
Empty queue: True
```

Let's practice!

DATA STRUCTURES AND ALGORITHMS IN PYTHON

Hash tables

DATA STRUCTURES AND ALGORITHMS IN PYTHON

Miriam Antona
Software engineer

datacamp

Definition

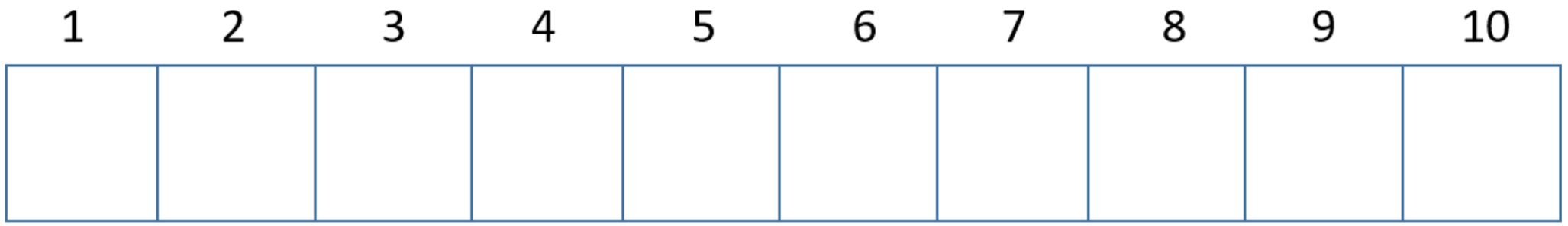
- Stores a collection of items
- **Key-value pairs**

```
lasagna: 14.75
moussaka: 21.15
sushi: 16.05
```

- Almost every programming language has a built-in hash table:
  - hashes, hash maps, dictionaries, associative arrays
  - Python: **dictionaries**

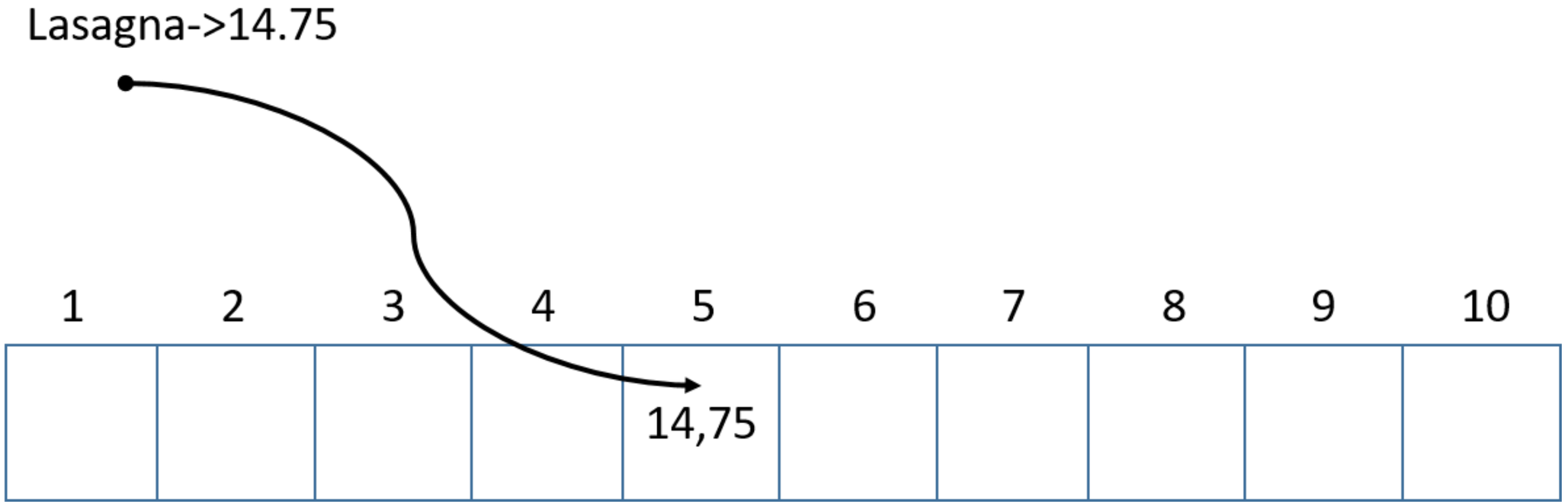
Structure

- Each position: **slot/bucket**

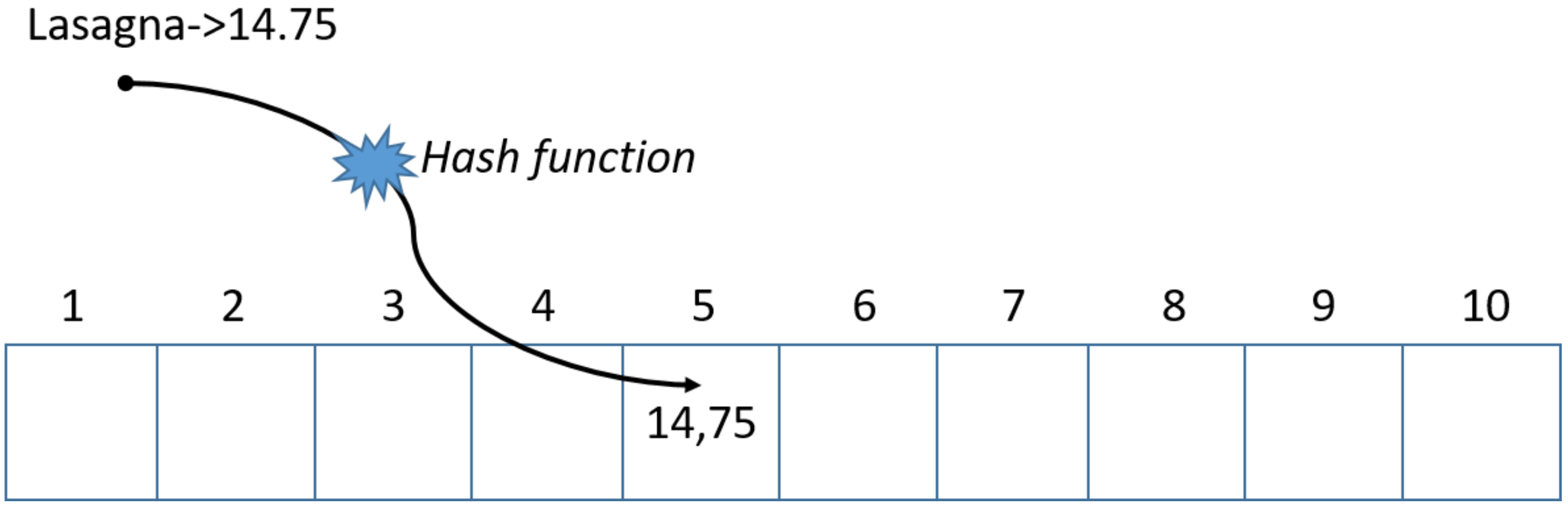


Empty slots

Hash functions

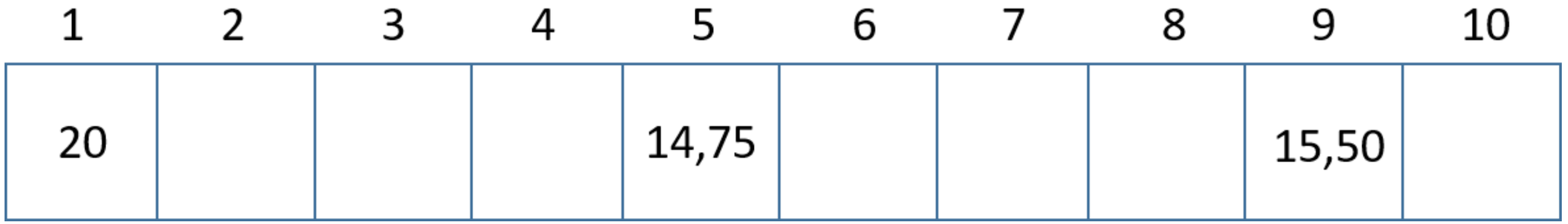


Hash functions



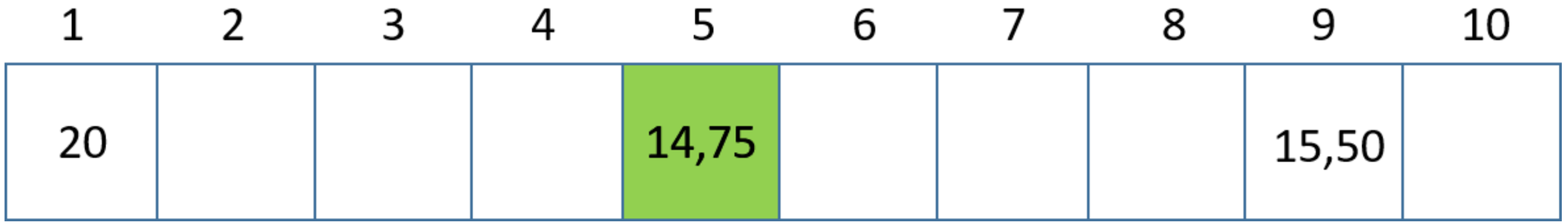
- Every time a hash function is applied
  - must return the **same value** for the **same input**

Lookups



- Find "lasagna"
  - hash("lasagna") -> 5

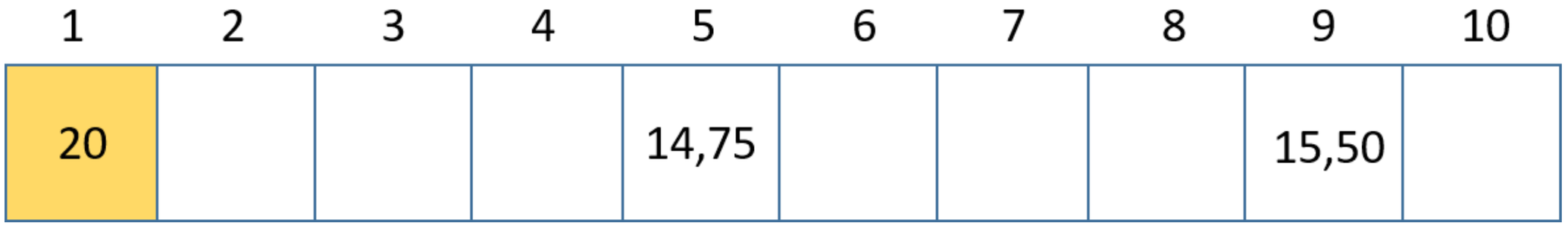
Lookups



- Find "lasagna"
  - hash("lasagna") -> 5
  - return 14,75
- $O(1)$

Collisions

- Hash functions can return **same output** for **different inputs**
- hash("moussaka") -> 1



- insert: moussaka -> 21,15
- **Collision!**
  - must be resolved
  - several techniques

Python dictionary

- Empty dictionary

```
my_empty_dictionary = {}
```

- Dictionary with items

```
my_menu = {
    'lasagna': 14.75,
    'moussaka': 21.15,
    'sushi': 16.05
}
```

Python dictionary - get

```
print(my_menu['sushi'])
```

```
16.05
```

```
print(my_menu['paella'])
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'paella'
```

```
print(my_menu.get('paella'))
```

```
None
```

Python dictionary - items, keys & values

- Get items

```
print(my_menu.items())
```

```
dict_items([('lasanga', 14.75),
            ('moussaka', 21.15),
            ('sushi', 16.05)])
```

- Get keys

```
print(my_menu.keys())
```

```
dict_keys(['lasanga', 'moussaka', 'sushi'])
```

- Get values

```
print(my_menu.values())
```

```
dict_values([14.75, 21.15, 16.05])
```

Python dictionary - insert

```
my_menu['samosas'] = 13
print(my_menu.items())
```

```
dict_items([('lasanga', 14.75), ('moussaka', 21.15), ('sushi', 16.05), ('samosas', 13)])
```

Python dictionary - modify

```
print(my_menu.get('sushi'))
```

```
16.05
```

```
my_menu['sushi'] = 20
print(my_menu.get('sushi'))
```

```
20
```

Python dictionary - remove

- Deletes a dictionary completely

```
del my_menu
```

- Removes a key-value pair

```
del my_menu["sushi"]
```

- Empties a dictionary

```
my_menu.clear()
```

Python dictionary - iterate

```
for key, value in my_menu.items():
  print(f"\nkey: {key}")
  print(f"value: {value}")
```

```
key: lasagna
value: 14.75

key: moussaka
value: 21.15

key: sushi
value: 16.05
```

Python dictionary - iterate

```
for dish in my_menu:
  print(dish)
```

```
lasagna
moussaka
sushi
```

```
for prices in my_menu.values():
  print(prices)
```

```
14.75
21.15
16.05
```

Python dictionary - nested dictionaries

- Nested dictionary

```
my_menu = {
  'sushi' : {
    'price' : 19.25,
    'best_served' : 'cold'
  },
  'paella' : {
    'price' : 15,
    'best_served' : 'hot'
  }
}
```

Let's practice!

DATA STRUCTURES AND ALGORITHMS IN PYTHON

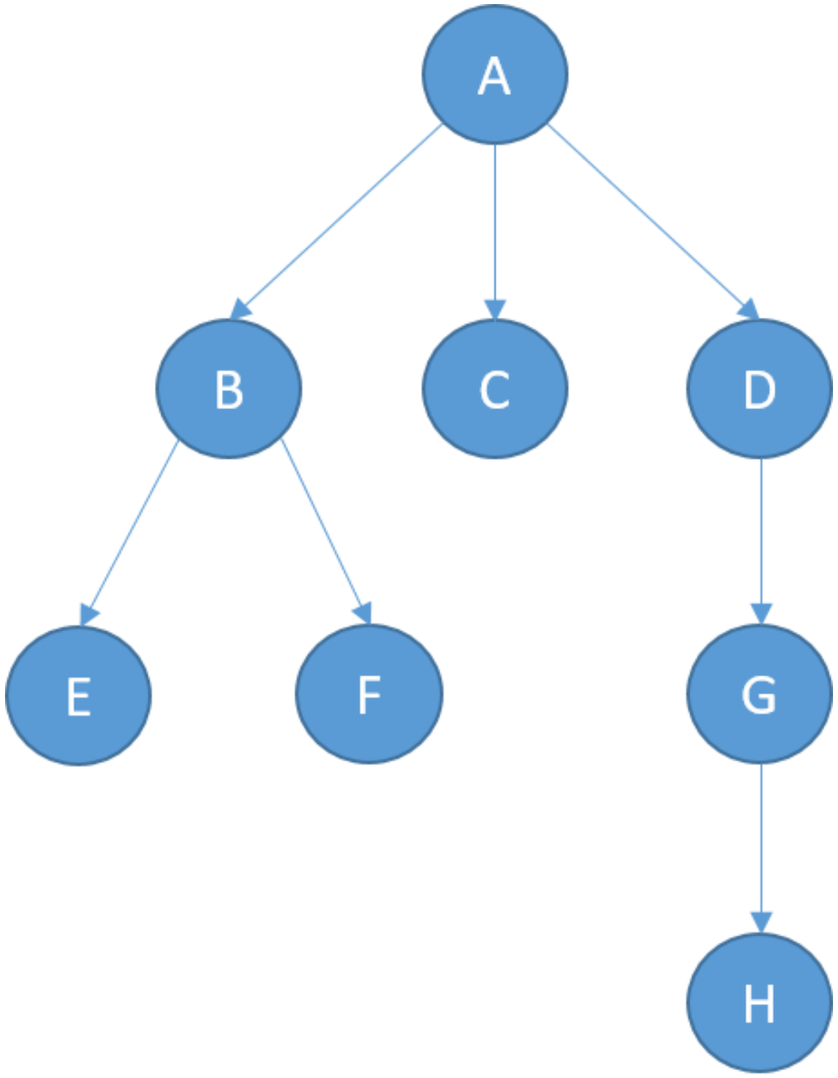
Trees and graphs

DATA STRUCTURES AND ALGORITHMS IN PYTHON

Miriam Antona
Software engineer

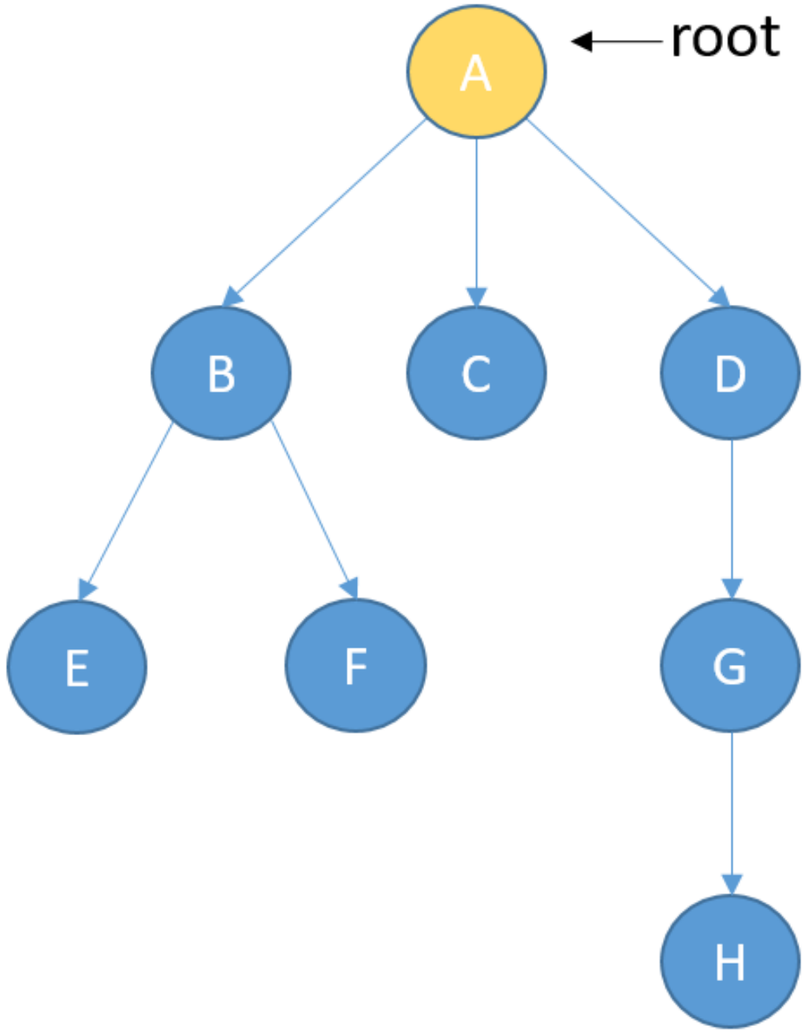
datacamp

Trees - definition

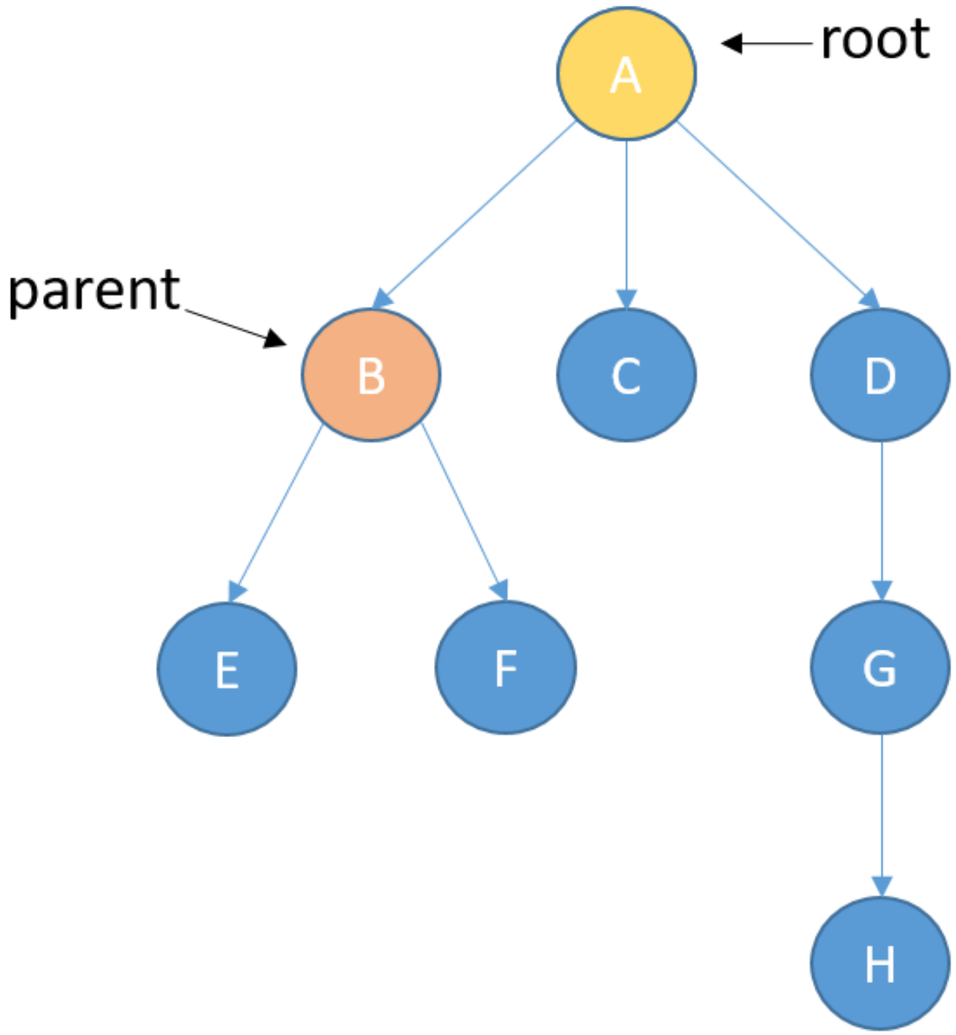


- **Node-based** data structures
- Each node can have **links** to **more than one node**.

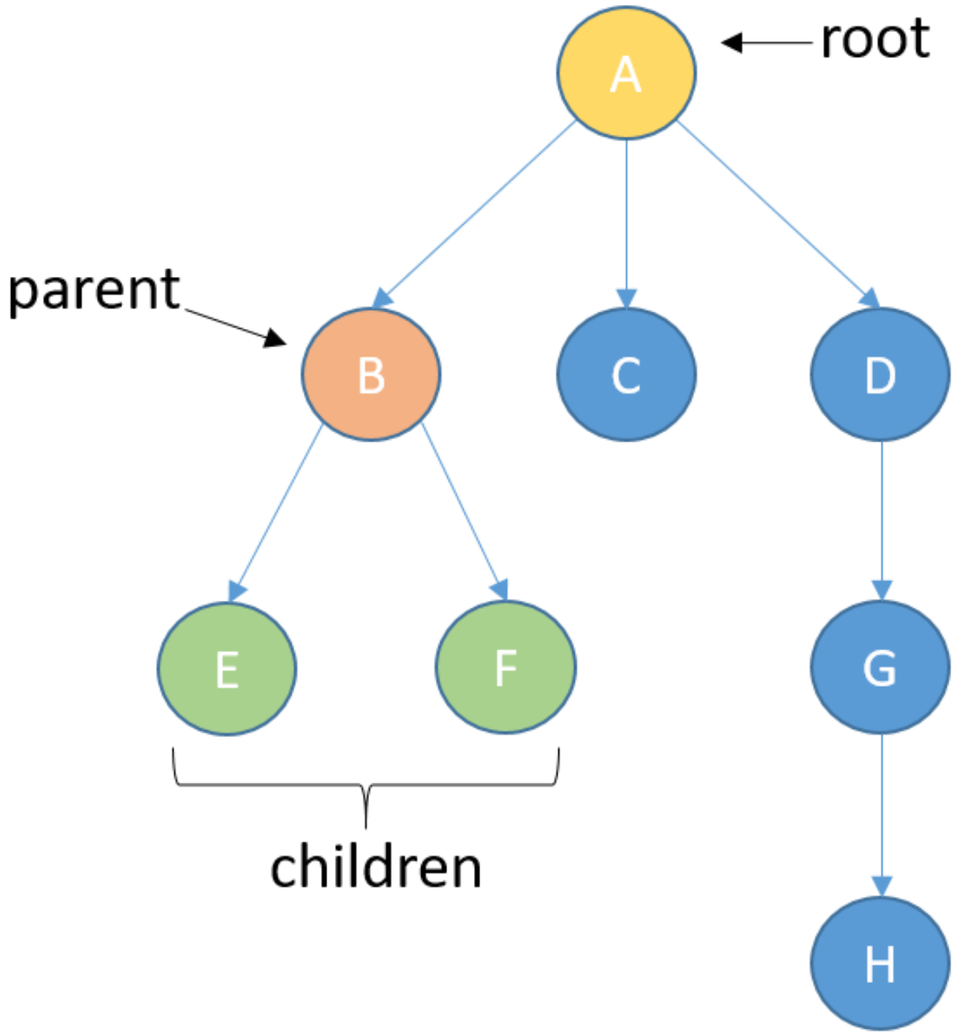
Trees - terminology



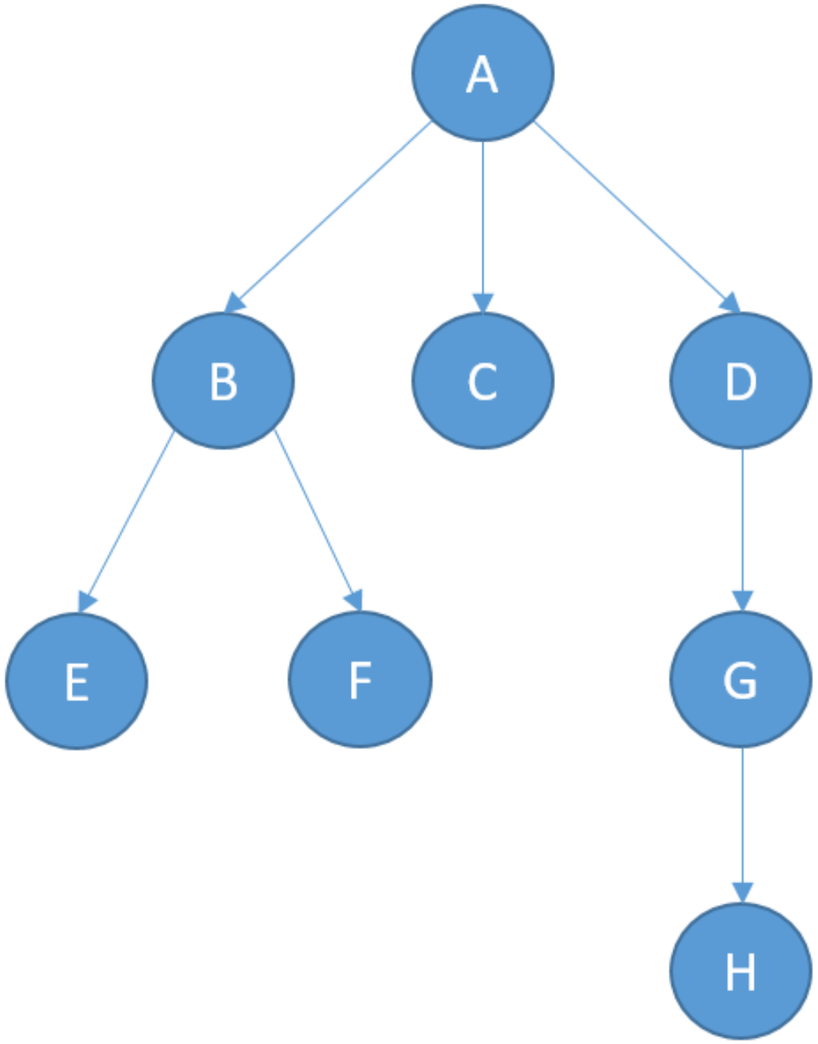
Trees - terminology



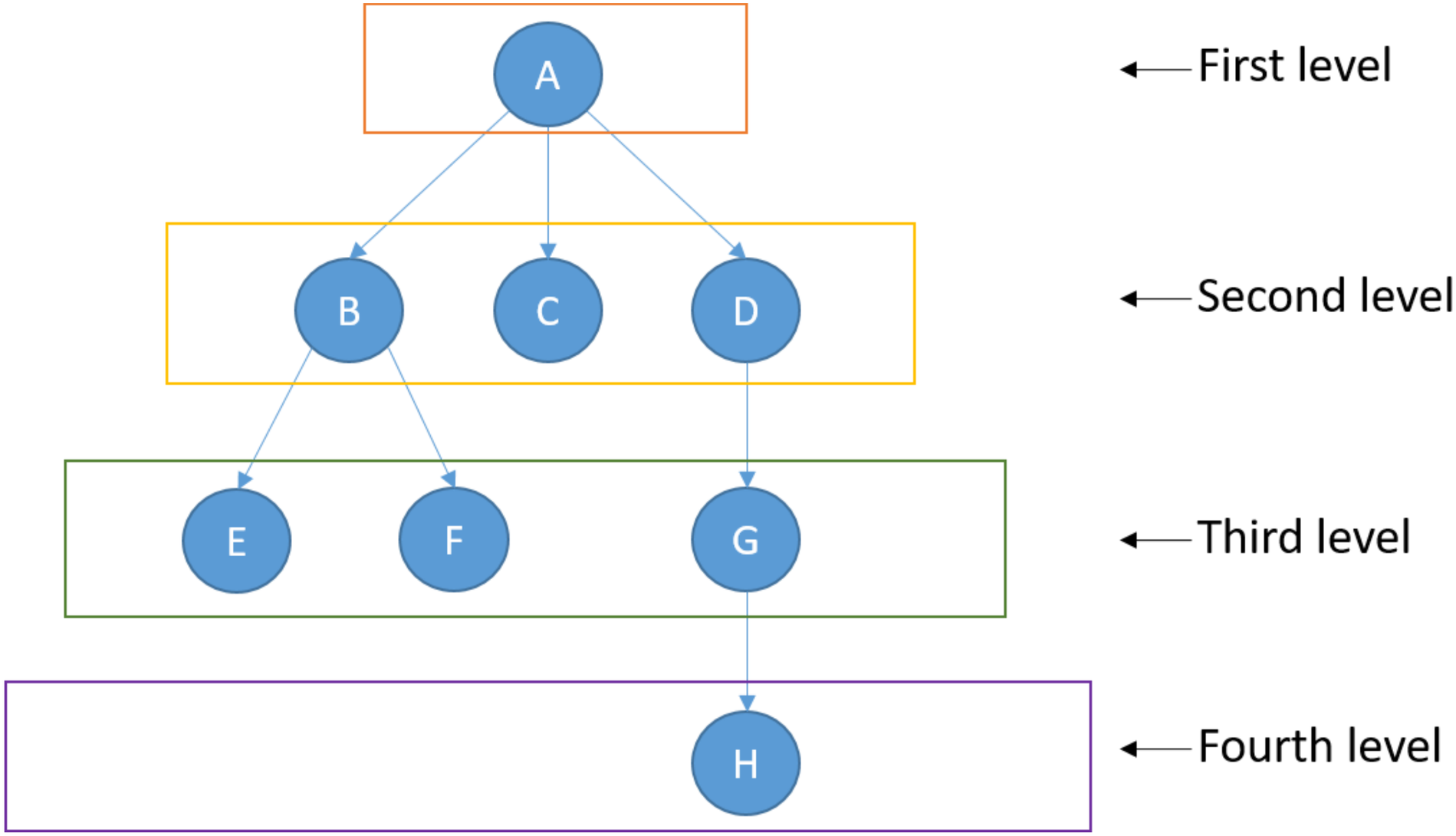
Trees - terminology



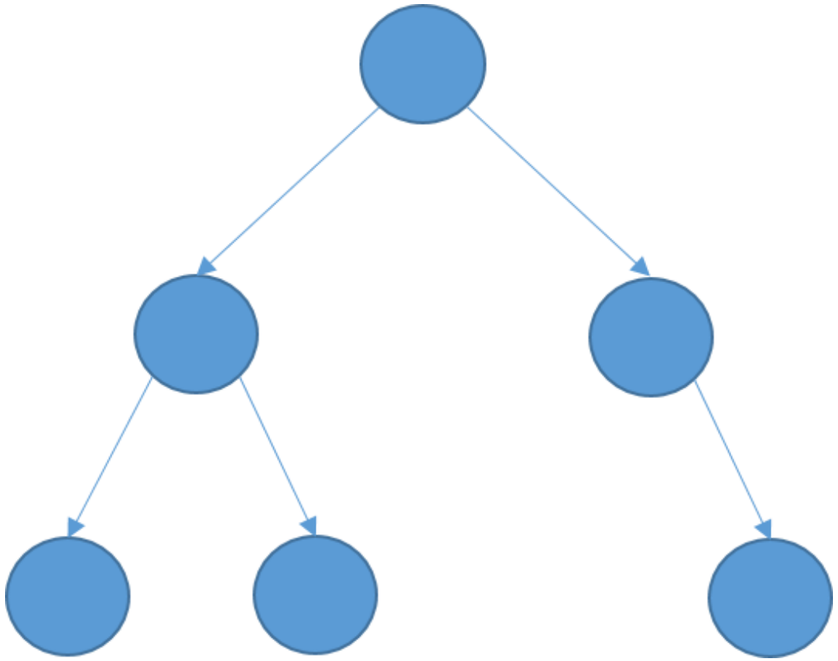
Trees - terminology



Trees - terminology



Trees - binary tree

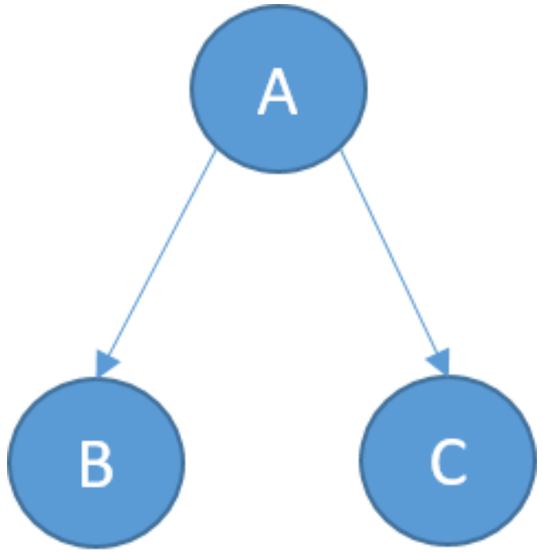


- Each node has:
  - zero children
  - one children
  - two children

Trees - binary tree implementation

```
class TreeNode:

  def __init__(self, data, left=None, right=None):
    self.data = data
    self.left_child = left
    self.right_child = right
```

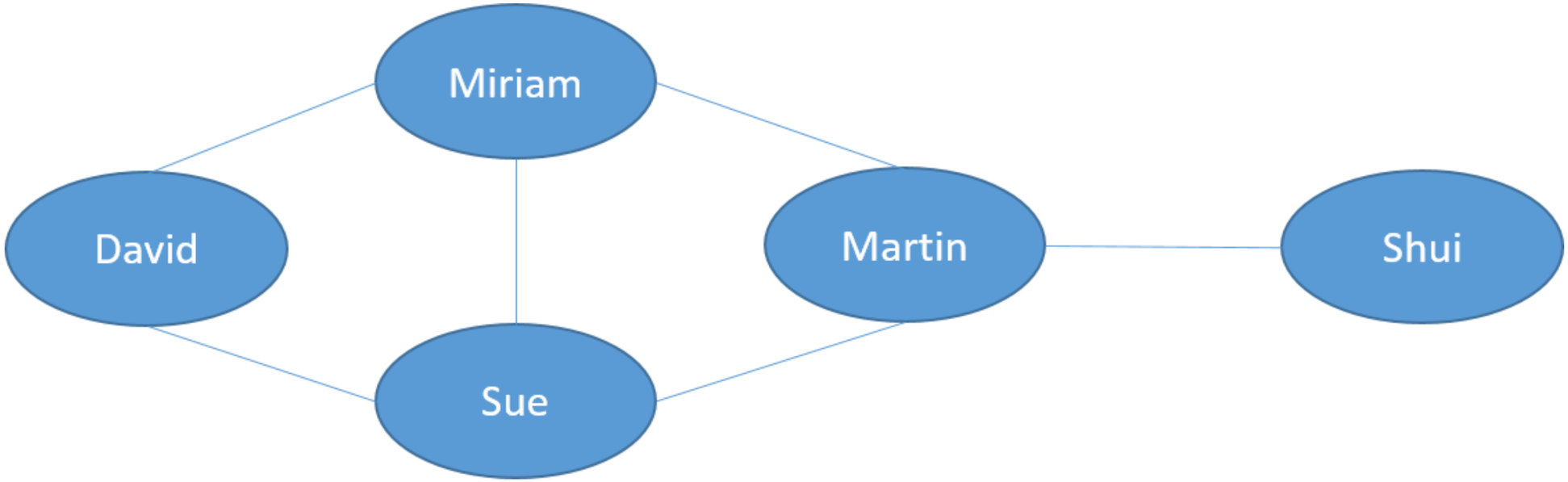



```
node1 = TreeNode("B")
node2 = TreeNode("C")
root_node = TreeNode("A", node1, node2)
```

Trees - real uses

- Storing **hierarchical relationships**
  - File system of a computer
  - Structure of an HTML document
- **Chess:** possible moves of the rival
- **Searching and sorting algorithms**

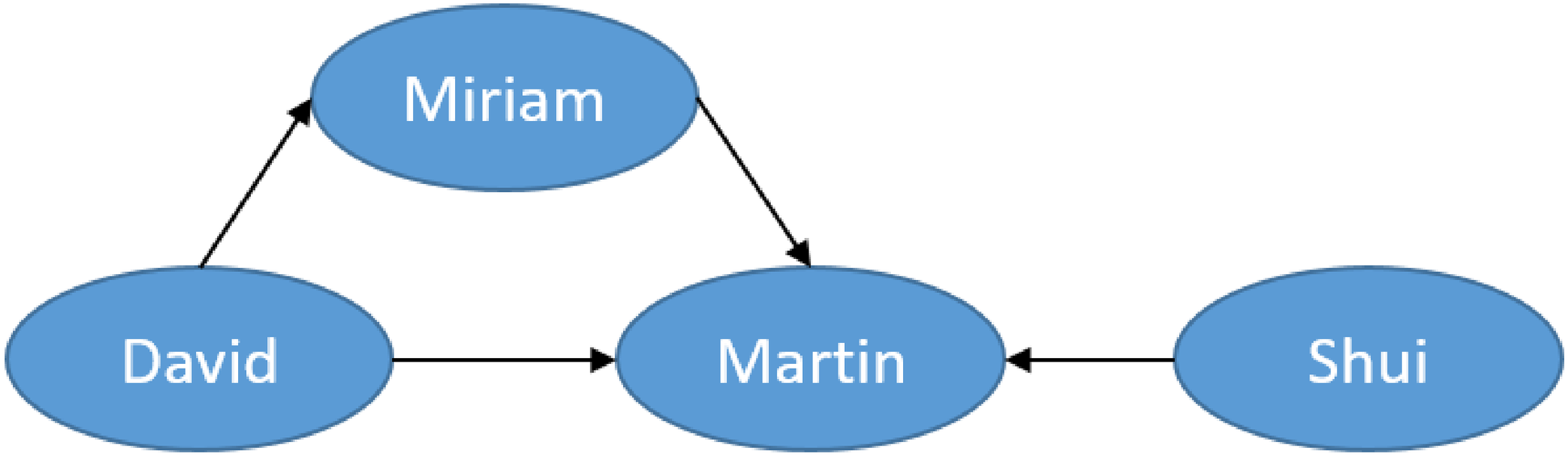
Graphs



- Set of:
  - nodes/vertices
  - links/edges
- Trees are a type of graph

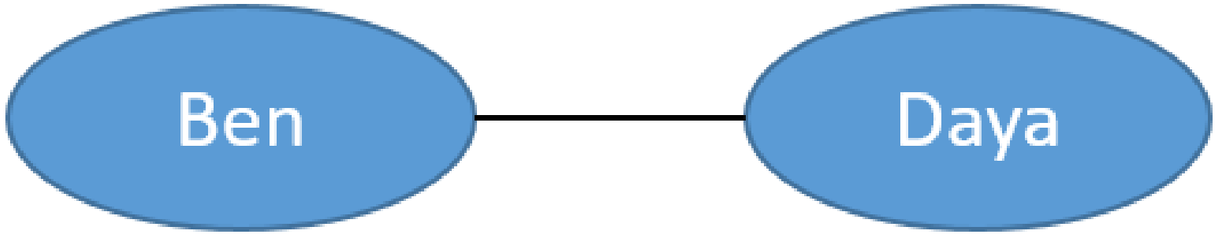
Graphs - types

- **Directed graphs:**
  - Specific direction



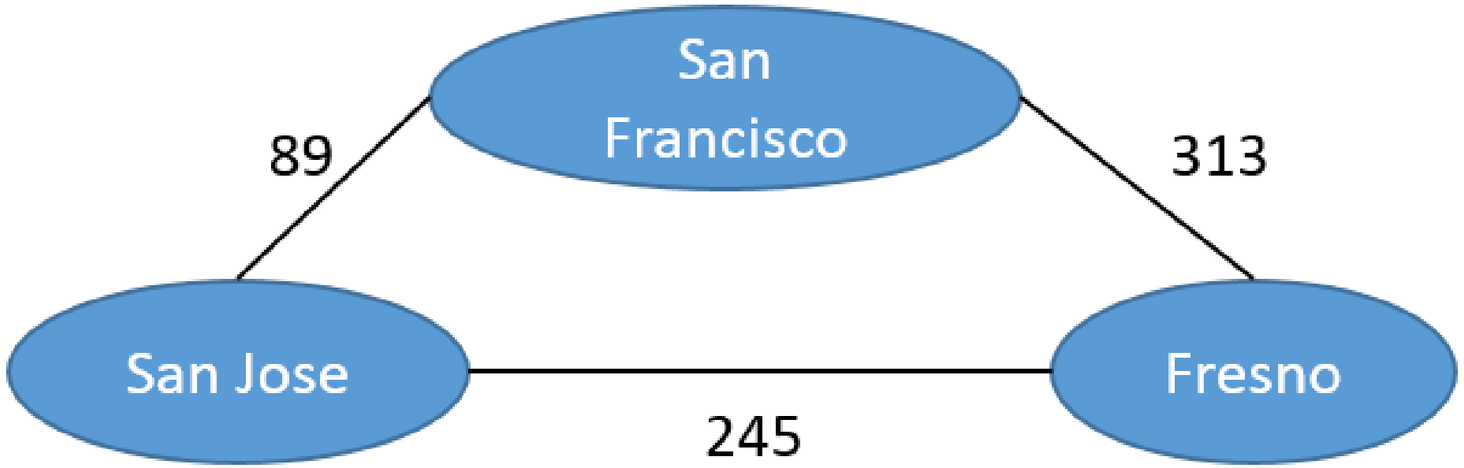
Graphs - types

- **Undirected graphs:**
  - Edges have no direction
  - The relationship is mutual



Graphs - types

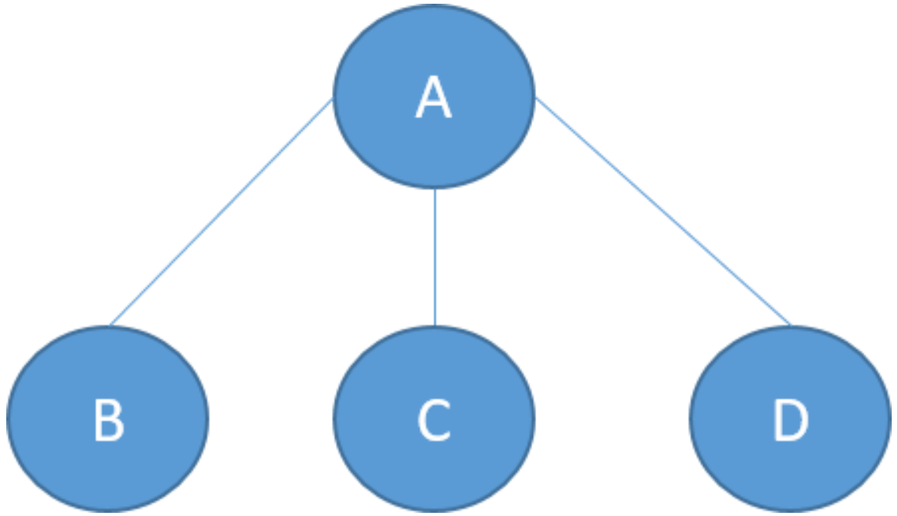
- **Weighted graphs:**
  - numeric values associated with the edges
  - can be either directed or undirected



Graphs vs. trees

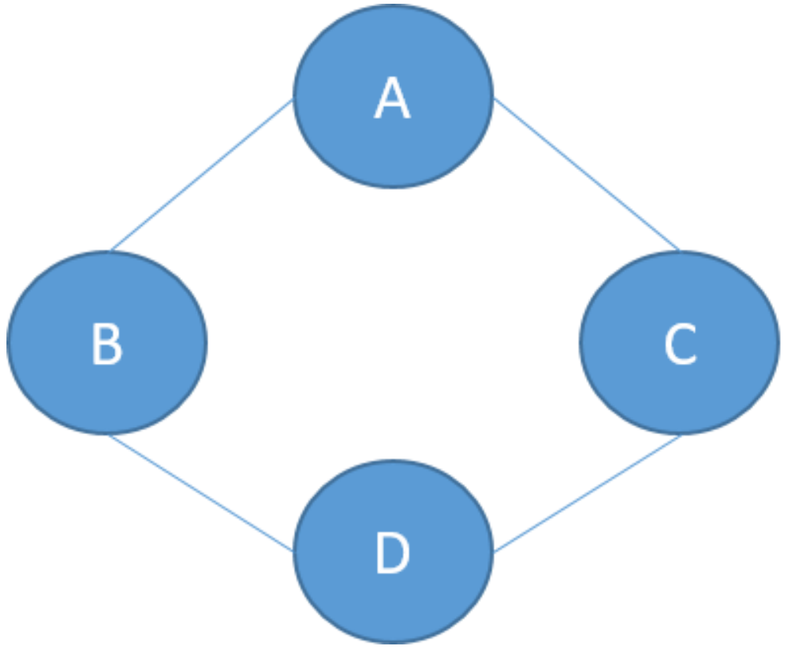
Trees

- Cannot have cycles
- All nodes must be connected



Graphs

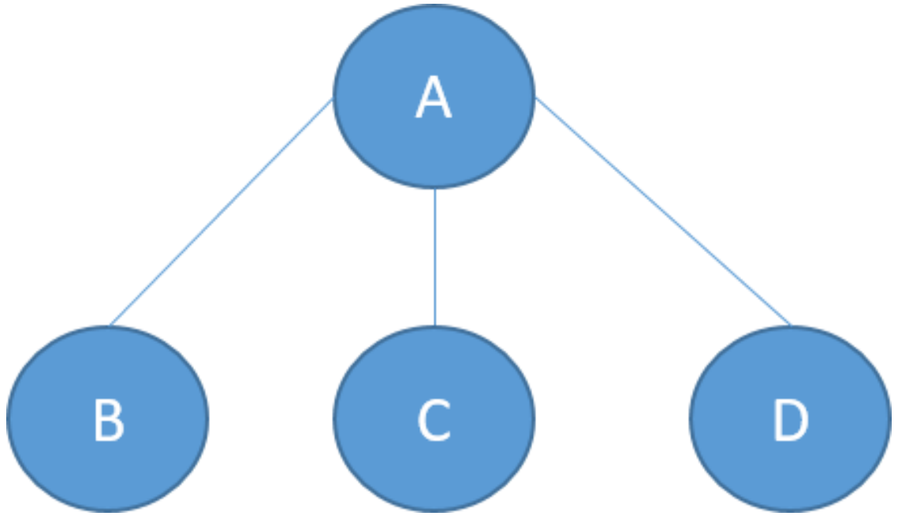
- Can have cycles
- There can be unconnected nodes



Graphs vs. trees

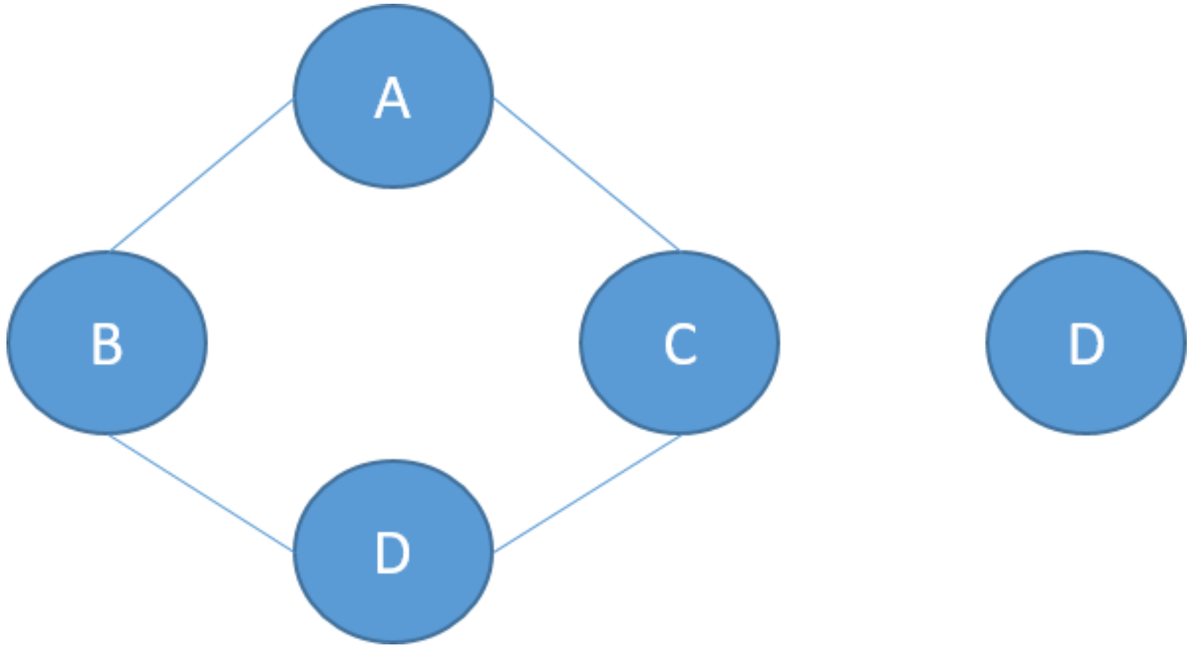
Trees

- Cannot have cycles
- All nodes must be connected



Graphs

- Can have cycles
- There can be unconnected nodes



Graphs - real world use-cases

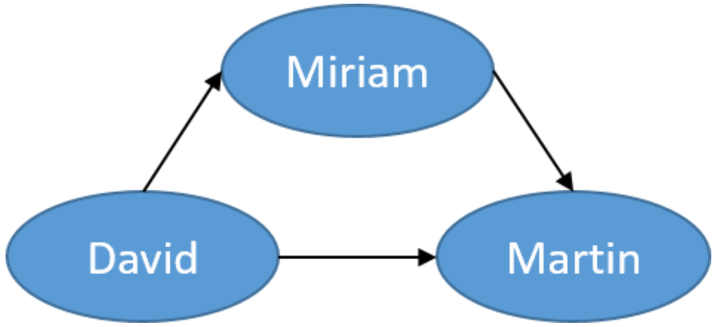
- User relationships in **social networks**
  - friendship
  - follows
  - likes
  - etc.
- **Locations and distances**
  - optimize routes
- **Graph databases**
- **Searching and sorting algorithms**

Graphs - implementation

```
class Graph:
  def__init(self):
    self.vertices = {}

  def add_vertex(self, vertex):
    self.vertices[vertex] = []

  def add_edge(self, source, target):
    self.vertices[source].append(target)
```




```
my_graph = Graph()
my_graph.add_vertex('David')
my_graph.add_vertex('Miriam')
my_graph.add_vertex('Martin')

my_graph.add_edge('David', 'Miriam')
my_graph.add_edge('David', 'Martin')
my_graph.add_edge('Miriam', 'Martin')
```

```
print(myGraph.vertices)
```

```
{
  'David' : ['Miriam','Martin'],
  'Miriam' : ['Martin'],
  'Martin' : []
}
```

Let's practice!

DATA STRUCTURES AND ALGORITHMS IN PYTHON

Understanding Recursion

DATA STRUCTURES AND ALGORITHMS IN PYTHON

Miriam Antona
Software engineer

datacamp

Definition

- Function calling itself
- Almost all the situations where we use loops
  - substitute the loops using recursion
- Can solve problems that seem very complex at first

Example - factorial

$n!$

Example - factorial

$$n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$$

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$$

```
def factorial(n):
  result = 1
  while n > 1:
    result = n * result
    n -= 1
  return result
```

```
factorial(5)
```

```
120
```

Example - factorial using recursion

$n! = n \cdot (n-1)!$

```
def factorial_recursion(n):
  return n * factorial_recursion(n-1)
```

- Executed forever!

Example - identifying the base case

- Add a condition
  - ensures that our algorithm doesn't execute forever
- **Factorial base case** -> $n = 1$

```
def factorial_recursion(n):
  if n == 1:
    return 1
  else:
    return n * factorial_recursion(n-1)
```

```
print(factorial_recursion(5))
```

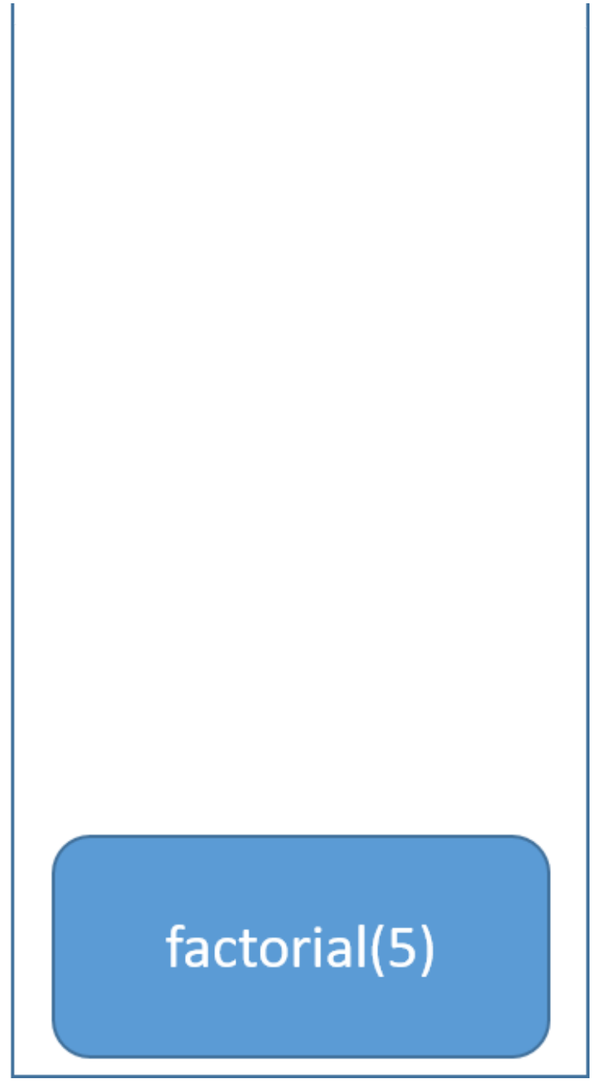
```
120
```

How recursion works

- Computer uses a **stack** to keep track of the functions
  - **Call stack**

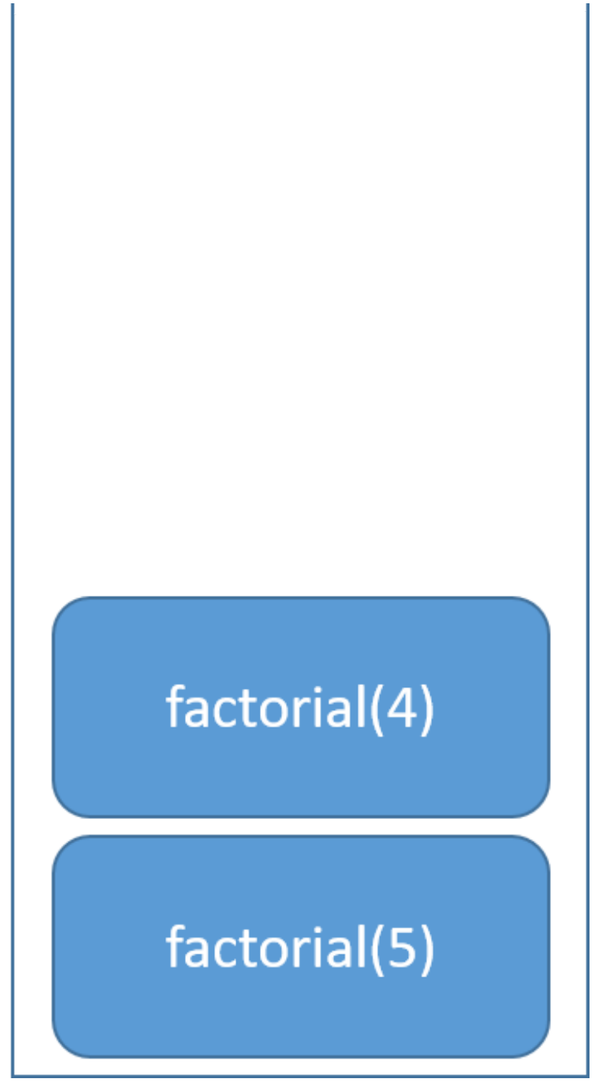
How recursion works

- `factorial(5)` starts
- Before `factorial(5)` finishes -> `factorial(4)` starts
- Before `factorial(4)` finishes -> `factorial(3)` starts



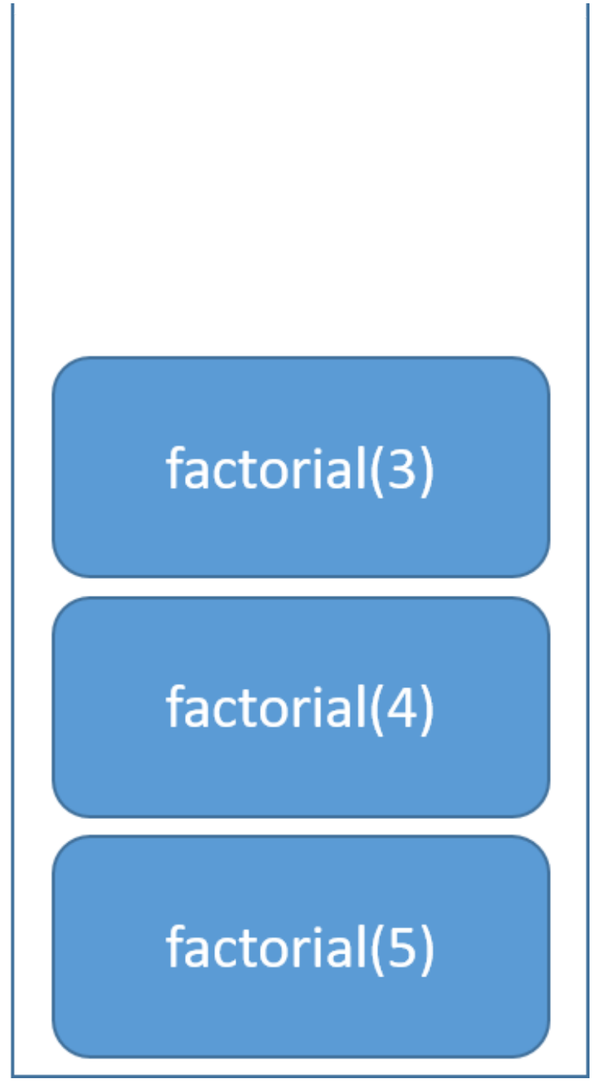
How recursion works

- `factorial(5)` starts
- Before `factorial(5)` finishes -> `factorial(4)` starts
- Before `factorial(4)` finishes -> `factorial(3)` starts
- Before `factorial(3)` finishes -> `factorial(2)` starts



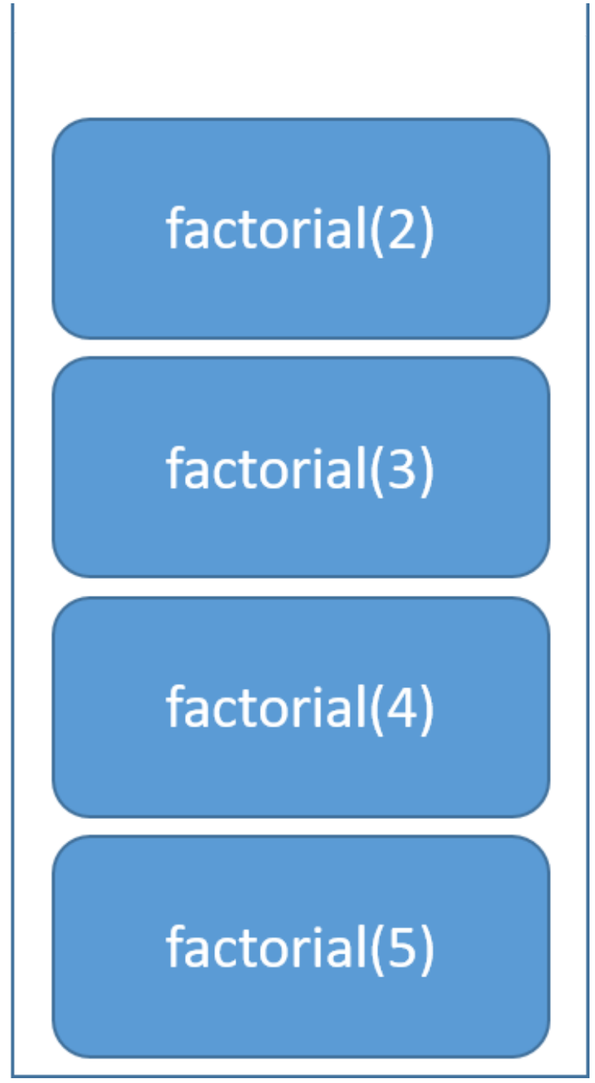
How recursion works

- `factorial(5)` starts
- Before `factorial(5)` finishes -> `factorial(4)` starts
- Before `factorial(4)` finishes -> `factorial(3)` starts
- Before `factorial(3)` finishes -> `factorial(2)` starts
- Before `factorial(2)` finishes -> `factorial(1)` starts



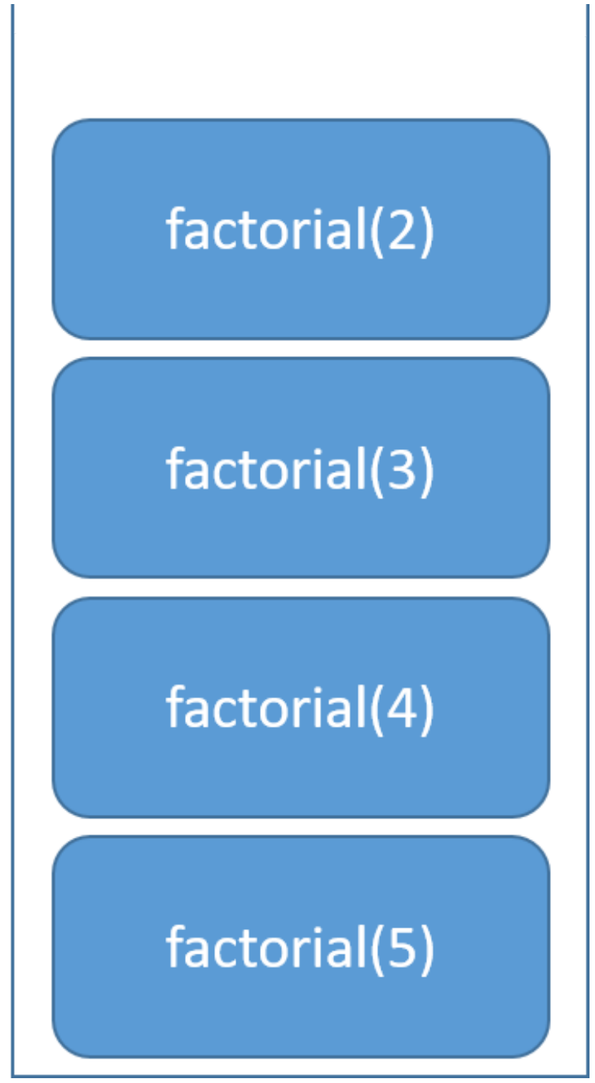
How recursion works

- `factorial(5)` starts
- Before `factorial(5)` finishes -> `factorial(4)` starts
- Before `factorial(4)` finishes -> `factorial(3)` starts
- Before `factorial(3)` finishes -> `factorial(2)` starts
- Before `factorial(2)` finishes -> `factorial(1)` starts



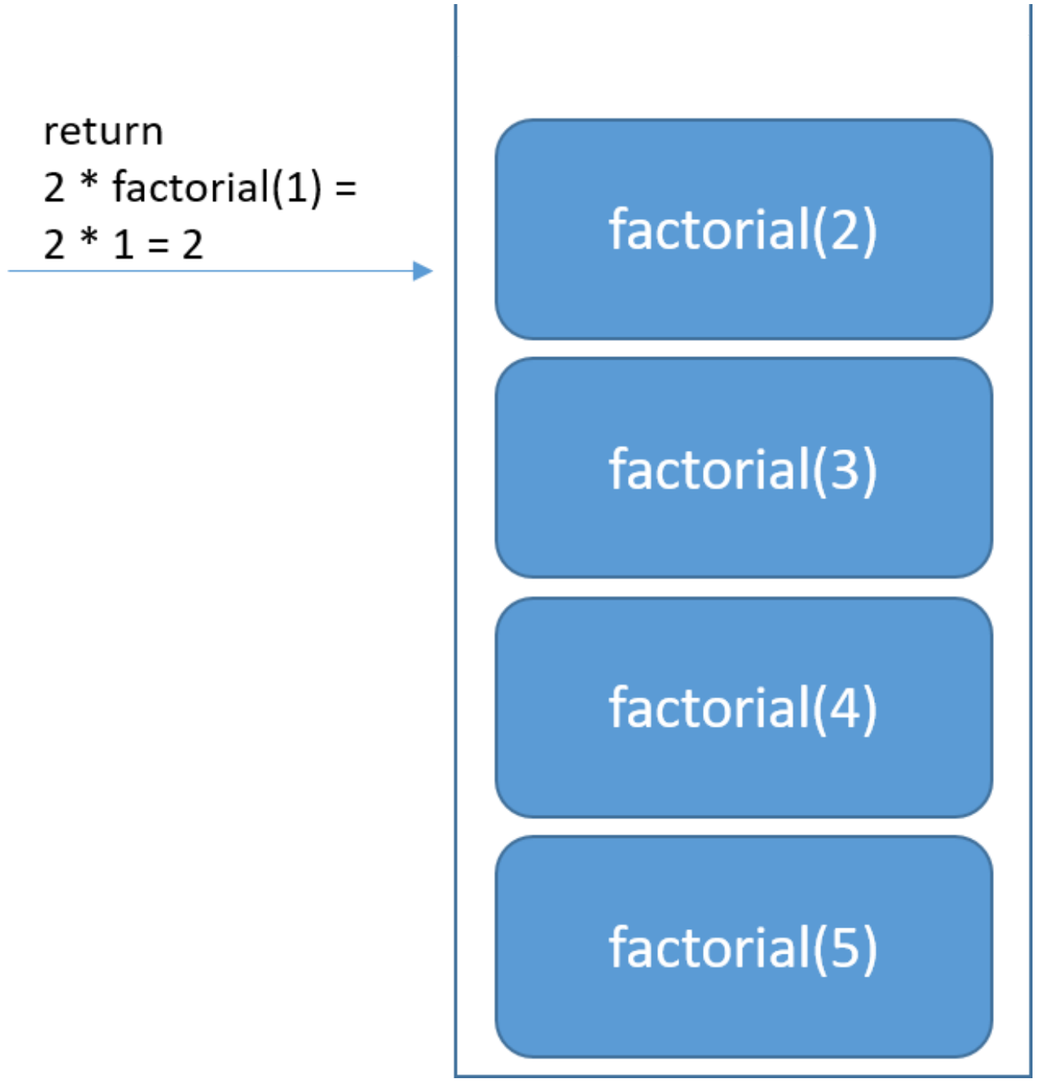
How recursion works

- `factorial(1)` finishes
  - returns 1
- `factorial(2)` finishes



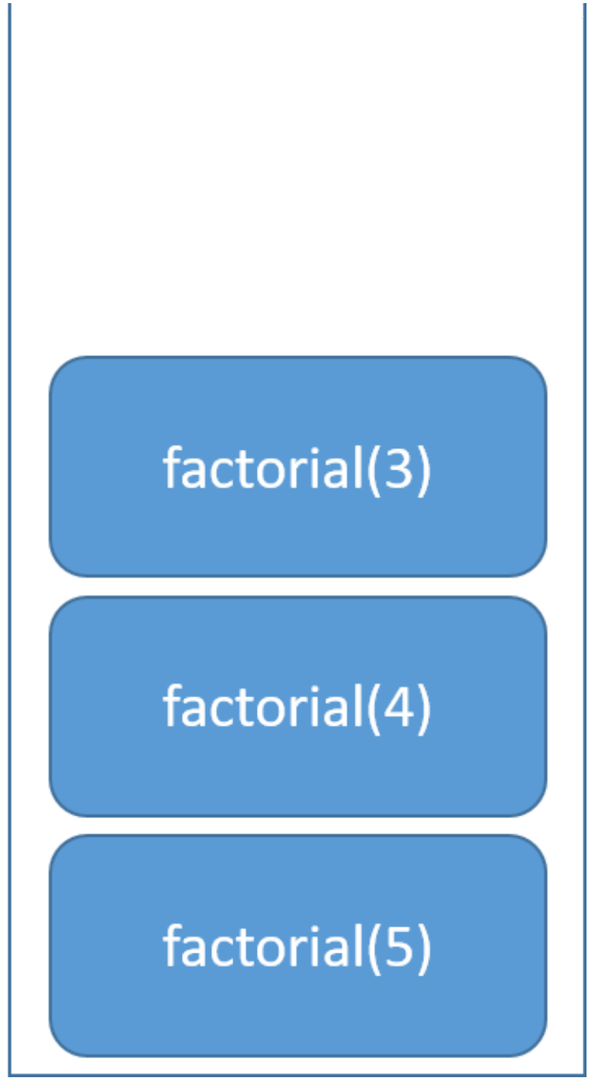
How recursion works

- `factorial(1)` finishes
  - returns 1
- `factorial(2)` finishes
  - returns 2



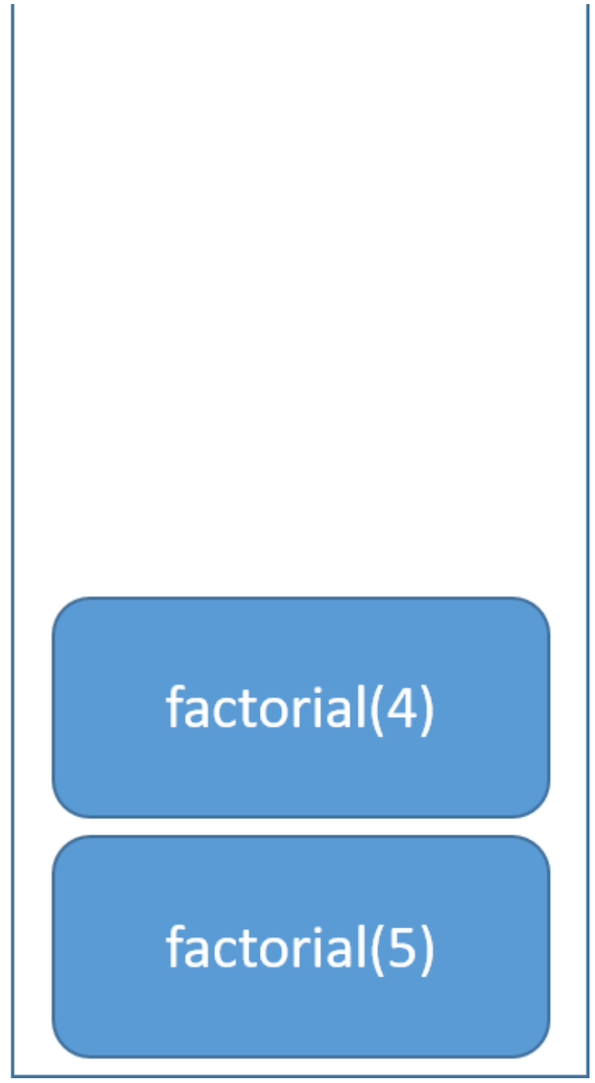
How recursion works

- `factorial(1)` finishes
  - returns 1
- `factorial(2)` finishes
  - returns 2
- `factorial(3)` finishes
  - returns 6



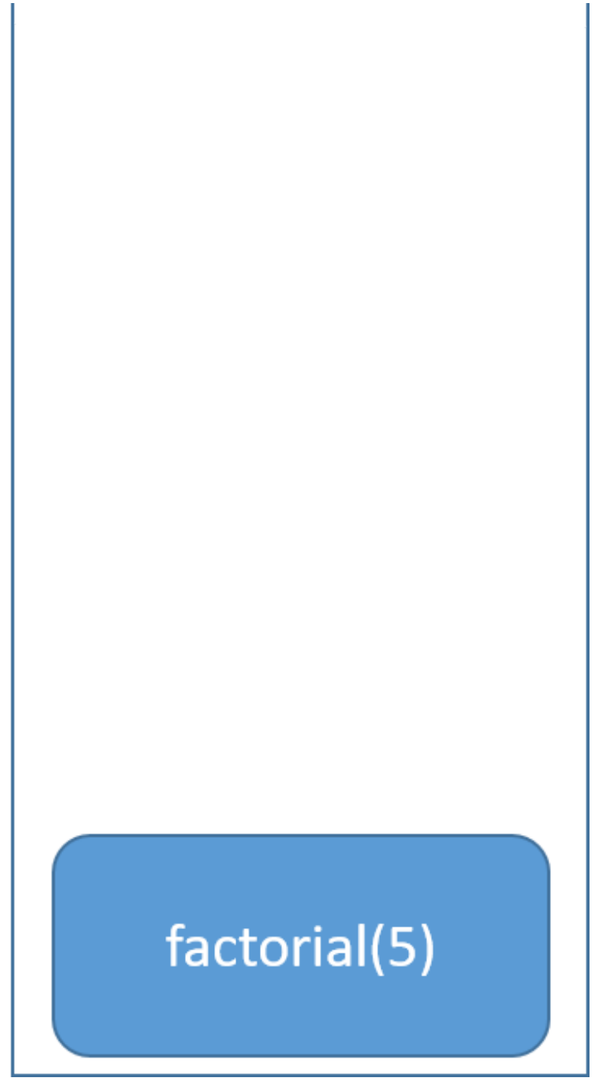
How recursion works

- `factorial(1)` finishes
  - returns 1
- `factorial(2)` finishes
  - returns 2
- `factorial(3)` finishes
  - returns 6
- `factorial(4)` finishes
  - returns 24



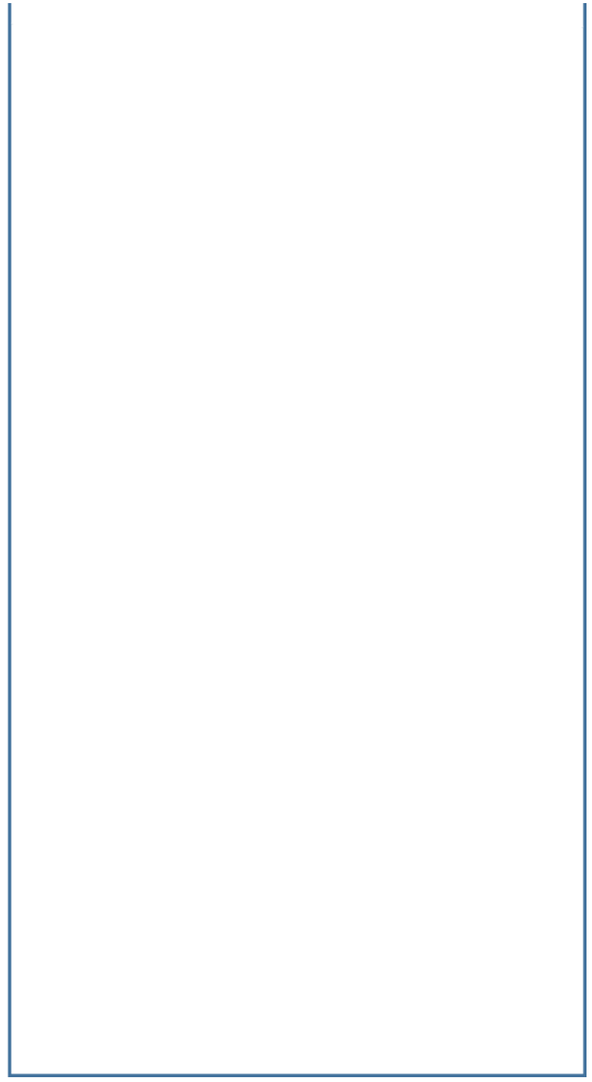
How recursion works

- `factorial(1)` finishes
  - returns 1
- `factorial(2)` finishes
  - returns 2
- `factorial(3)` finishes
  - returns 6
- `factorial(4)` finishes
  - returns 24
- `factorial(5)` finishes
  - returns 120



How recursion works

- `factorial(1)` finishes
  - returns 1
- `factorial(2)` finishes
  - returns 2
- `factorial(3)` finishes
  - returns 6
- `factorial(4)` finishes
  - returns 24
- `factorial(5)` finishes
  - returns 120



Dynamic programming

- Optimization technique
- Mainly applied to recursion
- Can reduce the complexity of recursive algorithms
- Used for:
  - Any problem that can be divided into smaller subproblems
  - Subproblems overlap
- Solutions of subproblems are saved, avoiding the need to recalculate
  - Memoization

Let's practice!

DATA STRUCTURES AND ALGORITHMS IN PYTHON