

# Welcome to Colab!

If you're already familiar with Colab, check out this video to learn about interactive tables, the executed code history view, and the command palette.



## What is Colab?

Colab, or "Colaboratory", allows you to write and execute Python in your browser, with

- Zero configuration required
- Access to GPUs free of charge
- Easy sharing

Whether you're a **student**, a **data scientist** or an **AI researcher**, Colab can make your work easier. Watch [Introduction to Colab](#) to learn more, or just get started below!

## ▼ Getting started

The document you are reading is not a static web page, but an interactive environment called a **Colab notebook** that lets you write and execute code.

For example, here is a **code cell** with a short Python script that computes a value, stores it in a variable, and prints the result:

```
seconds_in_a_day = 24 * 60 * 60
seconds_in_a_day
```

86400

To execute the code in the above cell, select it with a click and then either press the play button to the left of the code, or use the keyboard shortcut "Command/Ctrl+Enter". To edit the code, just click the cell and start editing.

Variables that you define in one cell can later be used in other cells:

```
seconds_in_a_week = 7 * seconds_in_a_day
seconds_in_a_week
```

604800

Colab notebooks allow you to combine **executable code** and **rich text** in a single document, along with **images**, **HTML**, **LaTeX** and more. When you create your own Colab notebooks, they are stored in your Google Drive account. You can easily share your Colab notebooks with co-workers or friends, allowing them to comment on your notebooks or even edit them. To learn more, see [Overview of Colab](#). To create a new Colab notebook you can use the File menu above, or use the following link: [create a new Colab notebook](#).

Colab notebooks are Jupyter notebooks that are hosted by Colab. To learn more about the Jupyter project, see [jupyter.org](https://jupyter.org).

## ▼ Data science

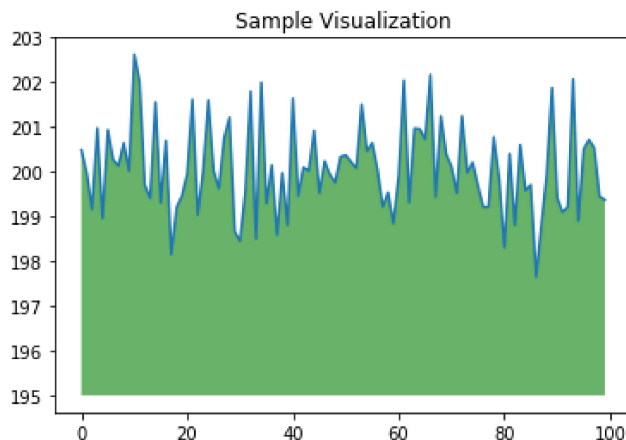
With Colab you can harness the full power of popular Python libraries to analyze and visualize data. The code cell below uses **numpy** to generate some random data, and uses **matplotlib** to visualize it. To edit the code, just click the cell and start editing.

```
import numpy as np
from matplotlib import pyplot as plt

ys = 200 + np.random.randn(100)
x = [x for x in range(len(ys))]

plt.plot(x, ys, '-')
plt.fill_between(x, ys, 195, where=(ys > 195), facecolor='g', alpha=0.6)

plt.title("Sample Visualization")
plt.show()
```



You can import your own data into Colab notebooks from your Google Drive account, including from spreadsheets, as well as from Github and many other sources. To learn more about importing data, and how Colab can be used for data science, see the links below under [Working with Data](#).

## ▼ Machine learning

With Colab you can import an image dataset, train an image classifier on it, and evaluate the model, all in just [a few lines of code](#). Colab notebooks execute code on Google's cloud servers, meaning you can leverage the power of Google hardware, including [GPUs and TPUs](#), regardless of the power of your machine. All you need is a browser.


Colab is used extensively in the machine learning community with applications including:

- Getting started with TensorFlow
- Developing and training neural networks
- Experimenting with TPUs
- Disseminating AI research
- Creating tutorials

To see sample Colab notebooks that demonstrate machine learning applications, see the [machine learning examples](#) below.

## ▼ More Resources

### Working with Notebooks in Colab

- [Overview of Colaboratory](#)
- [Guide to Markdown](#)
- [Importing libraries and installing dependencies](#)
- [Saving and loading notebooks in GitHub](#)
- [Interactive forms](#)
- [Interactive widgets](#)
- 

### Working with Data

- [Loading data: Drive, Sheets, and Google Cloud Storage](#)
- [Charts: visualizing data](#)
- [Getting started with BigQuery](#)

### Machine Learning Crash Course

These are a few of the notebooks from Google's online Machine Learning course. See the [full course website](#) for more.

- [Intro to Pandas DataFrame](#)
- [Linear regression with tf.keras using synthetic data](#)

### Using Accelerated Hardware

- [TensorFlow with GPUs](#)
- [TensorFlow with TPUs](#)

## ▼ Featured examples

- [NeMo Voice Swap](#): Use Nvidia's NeMo conversational AI Toolkit to swap a voice in an audio fragment with a computer generated one.
- [Retraining an Image Classifier](#): Build a Keras model on top of a pre-trained image classifier to distinguish flowers.

- [Text Classification](#): Classify IMDB movie reviews as either *positive* or *negative*.
- [Style Transfer](#): Use deep learning to transfer style between images.
- [Multilingual Universal Sentence Encoder Q&A](#): Use a machine learning model to answer questions from the SQuAD dataset.
- [Video Interpolation](#): Predict what happened in a video between the first and the last frame.

Name: Zahid Nawaz REG NO: FA20-BSE-021

Activity no.1

```
class node:
    def __init__(self,state,parent,actions,totalcost):
        self.state = state
        self.parent = parent
        self.actions = actions
        self.totalcost = totalcost

graph = {'A': node('A',None,['B','C','E'],None),
        'B': node('B',None,['A','D','E'],None),
        'C': node('C',None,['A','F','G'],None),
        'D': node('D',None,['B','E'],None),
        'E': node('E',None,['A','B','D'],None),
        'F': node('F',None,['C'],None),
        'G': node('G',None,['C'],None)
        }
```

Activity No.2

```
class node:
    def __init__(self,state,parent,actions,totalcost):
        self.state = state
        self.parent = parent
        self.actions = actions
        self.totalcost = totalcost

def actionSequence(graph,initialstate,goalstate):
    solution = [goalstate]
    currentparent = graph[goalstate].parent

    while currentparent != None:

        solution.append(currentparent)
        currentparent = graph[currentparent].parent

    solution.reverse()
    return solution

def dfs(initialstate,goalstate):

    graph = {'A': node('A',None,['B','C','E'],None),
```

```

        'B': node('B',None,['A','D','E'],None),
        'C': node('C',None,['A','F','G'],None),
        'D': node('D',None,['B','E'],None),
        'E': node('E',None,['A','B','D'],None),
        'F': node('F',None,['C'],None),
        'G': node('G',None,['C'],None)
    }
    frontier = [initialstate]
    explored = []
    currentChildren = 0
    while frontier:
        currentnode = frontier.pop(len(frontier)-1)
        explored.append(currentnode)
        for child in graph[currentnode].actions:
            if child not in frontier and child not in explored:
                graph[child].parent = currentnode
                if graph[child].state == goalstate:
                    # print(explored)
                    return actionSequence(graph,initialstate,goalstate)
                currentChildren=currentChildren+1
                frontier.append(child)
        if currentChildren == 0 :
            del explored[len(explored)-1]
    solution = dfs('A','D')
    print(solution)

    ['A', 'E', 'D']

```

### Activity No.3

```

class node:
    def __init__(self,state,parent,actions,totalcost):
        self.state = state
        self.parent = parent
        self.actions = actions
        self.totalcost = totalcost

def actionSequence(graph,initialstate,goalstate):
    solution = [goalstate]
    currentparent = graph[goalstate].parent

    while currentparent != None:

        solution.append(currentparent)
        currentparent = graph[currentparent].parent

    solution.reverse()
    return solution

def bfs(initialstate,goalstate):

    graph = {'A': node('A',None,['B','C','E'],None),
            'B': node('B',None,['A','D','E'],None),
            'C': node('C',None,['A','F','G'],None),
            'D': node('D',None,['B','E'],None),
            'E': node('E',None,['A','B','D'],None),
            'F': node('F',None,['C'],None),
            'G': node('G',None,['C'],None)
    }

```

```

frontier = [initialstate]
explored = []
while frontier:
    currentnode = frontier.pop(0)
    explored.append(currentnode)
    for child in graph[currentnode].actions:
        if child not in frontier and child not in explored:
            graph[child].parent = currentnode
            if graph[child].state == goalstate:
                return actionSequence(graph,initialstate,goalstate)
            frontier.append(child)
solution = bfs('D','C')
print(solution)

['D', 'B', 'A', 'C']

```

#### Activity No.4

```

def dfs(initialstate,goalstate):

    graph = {'Oradea': node('Oradea',None,['Sibiu','Zerlind'],None),
             'Zerlind': node('Zerlind',None,['Arad','Oradea'],None),
             'Arad': node('Arad',None,['Sibiu','Timisoara','Zerlind'],None),
             'Timisoara': node('Timisoara',None,['Arad','Lugoj'],None),
             'Lugoj': node('Lugoj',None,['Mehadia','Timisoara'],None),
             'Mehadia': node('Mehadia',None,['Drobeta','Lugoj'],None),
             'Drobeta': node('Drobeta',None,['Craiova','Mehadia'],None),
             'Craiova': node('Craiova',None,['Drobeta','Pitesti','Rimnicu Vicea'],None),
             'Rimnicu Vicea': node('Rimnicu Vicea',None,['Craiova','Pitesti','Sibiu'],None),
             'Sibiu': node('Sibiu',None,['Arad','Oradea','Fagaras','Rimnicu Vicea'],None),
             'Fagaras': node('Fagaras',None,['Bucharest','Sibiu'],None),
             'Pitesti': node('Pitesti',None,['Bucharest','Craiova','Rimnicu Vicea'],None),
             'Bucharest': node('Bucharest',None,['Fagaras','Pitesti','Giurgiu','Urziceni'],None),
             'Giurgiu': node('Giurgiu',None,['Bucharest'],None),
             'Bucharest': node('Bucharest',None,['Fagaras','Pitesti','Giurgiu','Urziceni'],None),
             'Urziceni': node('Urziceni',None,['Bucharest','Hirsova','Vaslui'],None),
             'Hirsova': node('Hirsova',None,['Eforie','Urziceni'],None),
             'Eforie': node('Eforie',None,['Hirsova'],None),
             'Vaslui': node('Vaslui',None,['Iasi','Urziceni'],None),
             'Iasi': node('Iasi',None,['Neamt','Vaslui'],None),
             'Neamt': node('Neamt',None,['Iasi'],None),
            }

    frontier = [initialstate]
    explored = []
    while frontier:
        currentnode = frontier.pop()
        explored.append(currentnode)
        for child in reversed(graph[currentnode].actions):
            if child not in frontier and child not in explored:
                graph[child].parent = currentnode
                if graph[child].state == goalstate:
                    return actionSequence(graph,initialstate,goalstate)
                frontier.append(child)

    solution = dfs('Arad','Bucharest')
    print(solution)

```

```
['Arad', 'Sibiu', 'Fagaras', 'Bucharest']
```

Double-click (or enter) to edit

#### Activity no.4

```
from queue import PriorityQueue

# Define the tree as a dictionary of dictionaries
tree = {'A': {'B': 1, 'C': 3},
        'B': {'D': 2, 'E': 1},
        'C': {'F': 4},
        'D': {},
        'E': {'G': 3},
        'F': {},
        'G': {}}

def uniform_cost_search(tree, start, goal):
    frontier = PriorityQueue()
    frontier.put((0, start))
    explored = []
    path = {}
    path[start] = None

    while not frontier.empty():
        cost, current_node = frontier.get()
        explored.append(current_node)

        if current_node == goal:
            final_path = []
            while current_node in path:
                final_path.append(current_node)
                current_node = path[current_node]
            final_path.reverse()
            return final_path

        for neighbor, neighbor_cost in tree[current_node].items():
            if neighbor not in explored:
                new_cost = cost + neighbor_cost
                if neighbor not in [node[1] for node in frontier.queue]:
                    frontier.put((new_cost, neighbor))
                    path[neighbor] = current_node
                elif new_cost < [node[0] for node in frontier.queue if node[1] == neighbor][0]:
                    frontier.get([node for node in frontier.queue if node[1] == neighbor][0])
                    frontier.put((new_cost, neighbor))
                    path[neighbor] = current_node

    return None

# Test the uniform cost search algorithm
start_node = 'C'
goal_node = 'B'
result_path = uniform_cost_search(tree, start_node, goal_node)

if result_path:
    print("The minimum cost path from", start_node, "to", goal_node, "is:")
    print(result_path)
```

```

    print("The total cost is:", sum(tree[result_path[i]][result_path[i+1]] for i in range(len(result_path)-1))
else:
    print("Goal not reachable from the starting node")

    Goal not reachable from the starting node

```

## Activity no.5

```

import heapq

# Define the graph as a dictionary
graph = {
    'Arad': [('Zerind', 75), ('Timisoara', 118), ('Sibiu', 140)],
    'Zerind': [('Oradea', 71), ('Arad', 75)],
    'Oradea': [('Sibiu', 151), ('Zerind', 71)],
    'Timisoara': [('Arad', 118), ('Lugoj', 111)],
    'Lugoj': [('Timisoara', 111), ('Mehadia', 70)],
    'Mehadia': [('Lugoj', 70), ('Drobeta', 75)],
    'Drobeta': [('Mehadia', 75), ('Craiova', 120)],
    'Sibiu': [('Arad', 140), ('Oradea', 151), ('Fagaras', 99), ('Rimnicu Vilcea', 80)],
    'Fagaras': [('Sibiu', 99), ('Bucharest', 211)],
    'Rimnicu Vilcea': [('Sibiu', 80), ('Craiova', 146), ('Pitesti', 97)],
    'Craiova': [('Drobeta', 120), ('Rimnicu Vilcea', 146), ('Pitesti', 138)],
    'Pitesti': [('Rimnicu Vilcea', 97), ('Craiova', 138), ('Bucharest', 101)],
    'Bucharest': [('Fagaras', 211), ('Pitesti', 101)]
}

def uniform_cost_search(start, goal):
    # Keep track of visited nodes and their distances from the start node
    visited = {start: 0}
    # Keep track of the nodes in the path from the start node to the current node
    path = {start: [start]}
    # Initialize the heap with the start node and its cost
    heap = [(0, start)]

    while heap:
        # Pop the node with the lowest cost from the heap
        (cost, current) = heapq.heappop(heap)

        # If we have reached the goal node, return the path
        if current == goal:
            return path[current]

        # Loop through the neighboring nodes
        for (neighbor, neighbor_cost) in graph[current]:
            # Calculate the new cost to reach the neighboring node
            new_cost = visited[current] + neighbor_cost

            # If the neighboring node hasn't been visited yet or the new cost is lower than the current cost
            if neighbor not in visited or new_cost < visited[neighbor]:
                # Update the visited dictionary and the path dictionary
                visited[neighbor] = new_cost
                path[neighbor] = path[current] + [neighbor]
                # Add the neighboring node and its cost to the heap
                heapq.heappush(heap, (new_cost, neighbor))

    return None

```



```

start = 'Arad'
goal = 'Bucharest'
path = uniform_cost_search(start, goal)
print(path)

```

```
['Arad', 'Sibiu', 'Rimnicu Vilcea', 'Pitesti', 'Bucharest']
```

```
class Node:
```

```

    def __init__(self, state, parent, actions, totalcost):
        self.state=state
        self.parent=parent
        self.actions=actions
        self.totalcost=totalcost

```

```

graph={'arad':Node('arad',None,['zernid','timisoara','sibiu'],None),
      'timisoara':Node('timisoara',None,['lugoj','arad'],None),
      'zernid':Node('zernid',None,['arad','oradea'],None),
      'sibiu':Node('sibiu',None,['arad','oradea','fagaras','rimnicu vilcea'],None),
      'lugoj':Node('lugoj',None,['mehadia','timisoara'],None),
      'oradea':Node('oradea',None,['zernid','sibiu'],None),
      'mehadia':Node('mehadia',None,['lugoj','drobeta'],None),
      'drobeta':Node('drobeta',None,['mehadia','craiova'],None),
      'craiova':Node('craiova',None,['drobeta','pitesti','rimnicu vilcea'],None),
      'rimnicu vilcea':Node('rimnicu vilcea',None,['craiova','pitesti','sibiu'],None),
      'pitesti':Node('pitesti',None,['craiova','rimnicu vilcea','bucharest'],None),
      'fagaras':Node('fagaras',None,['sibiu','bucharest'],None),
      'bucharest':Node('bucharest',None,['fagaras','pitesti','giurgiu','urziceni'],None),
      'giurgiu':Node('giurgiu',None,['bucharest'],None),
      'urziceni':Node('urziceni',None,['bucharest','hirsova','vaslui'],None),
      'hirsova':Node('hirsova',None,['urziceni','eforie'],None),
      'eforie':Node('eforie',None,['hirsova'],None),
      'vaslui':Node('vaslui',None,['urziceni','lasi'],None),
      'lasi':Node('lasi',None,['vaslui','neamt'],None),
      'neamt':Node('neamt',None,['lasi'],None),
}

```

```

def actionsequence(graph, initialstate,goalstate):
    solution=[goalstate]
    currentparent = graph[goalstate].parent
    while currentparent != None:
        solution.append(currentparent)
        currentparent = graph[currentparent].parent
    solution.reverse()
    print(solution)
    return solution

```

```
def BFS():
```

```

    initialstate = 'arad'
    goalstate = 'bucharest'
    frontier = [initialstate]
    explored = []

```

```

while len(frontier)!=0:
    currentNode = frontier.pop(len(frontier)-1)
    explored.append(currentNode)
    for child in graph[currentNode].actions:
        if child not in frontier and child not in explored:
            graph[child].parent = currentNode

```

```

        if graph[child].state == goalstate:
            return actionsequence(graph,initialstate,goalstate)
        frontier.append(child)
solution = BFS()

class Node:
    def __init__(self,state,parent,actions,totalcost):
        self.state=state
        self.parent=parent
        self.actions=actions
        self.totalcost=totalcost

graph={'arad':Node('arad',None,['zernid','timisoara','sibiu'],None),
      'timisoara':Node('timisoara',None,['lugoj','arad'],None),
      'zernid':Node('zernid',None,['arad','oradea'],None),
      'sibiu':Node('sibiu',None,['arad','oradea','fagaras','rimnicu vilcea'],None),
      'lugoj':Node('lugoj',None,['mehadia','timisoara'],None),
      'oradea':Node('oradea',None,['zernid','sibiu'],None),
      'mehadia':Node('mehadia',None,['lugoj','drobeta'],None),
      'drobeta':Node('drobeta',None,['mehadia','craiova'],None),
      'craiova':Node('craiova',None,['drobeta','pitesti','rimnicu vilcea'],None),
      'rimnicu vilcea':Node('rimnicu vilcea',None,['craiova','pitesti','sibiu'],None),
      'pitesti':Node('pitesti',None,['craiova','rimnicu vilcea','bucharest'],None),
      'fagaras':Node('fagaras',None,['sibiu','bucharest'],None),
      'bucharest':Node('bucharest',None,['fagaras','pitesti','giurgiu','urziceni'],None),
      'giurgiu':Node('giurgiu',None,['bucharest'],None),
      'urziceni':Node('urziceni',None,['bucharest','hirsova','vaslui'],None),
      'hirsova':Node('hirsova',None,['urziceni','eforie'],None),
      'eforie':Node('eforie',None,['hirsova'],None),
      'vaslui':Node('vaslui',None,['urziceni','lasi'],None),
      'lasi':Node('lasi',None,['vaslui','neamt'],None),
      'neamt':Node('neamt',None,['lasi'],None),
}

def actionsequence(graph, initialstate,goalstate):
    solution=[goalstate]
    currentparent = graph[goalstate].parent
    while currentparent != None:
        solution.append(currentparent)
        currentparent = graph[currentparent].parent
    solution.reverse()
    print(solution)
    return solution


def BFS():
    initialstate = 'arad'
    goalstate = 'bucharest'
    frontier = [initialstate]
    explored = []

    while len(frontier)!=0:
        currentNode = frontier.pop(len(frontier)-1)
        explored.append(currentNode)
        for child in graph[currentNode].actions:
            if child not in frontier and child not in explored:
                graph[child].parent = currentNode
                if graph[child].state == goalstate:
                    return actionsequence(graph,initialstate,goalstate)
                frontier.append(child)

```

```
solution = BFS()
```

---

 9m 39s    completed at 4:08 AM



Could not connect to the reCAPTCHA service. Please check your internet connection and reload to get a reCAPTCHA challenge.