Elementary Introduction to Artificial Intelligence and Machine Learning

Ilya Feldshteyn

2018

Table of Contents

[Chapter 2 Simple decisions 5](#_Toc521520980)

[2.1 Finding your path 5](#_Toc521520981)

[2.2 How to navigate a tree 7](#_Toc521520982)

[2.3 How to describe a tree 8](#_Toc521520983)

[2.4 Recursion 10](#_Toc521520984)

[2.5 Navigating a tree (depth first) 11](#_Toc521520985)

[2.6 Additional Exercises 12](#_Toc521520986)

[Chapter 3 Labyrinths 14](#_Toc521520987)

[3.1 A maze as a tree 14](#_Toc521520988)

[3.2 Maze descriptions 15](#_Toc521520989)

[3.3 Converting a maze to a tree. Graphs. 16](#_Toc521520990)

[3.4 Finding the exit 18](#_Toc521520991)

[3.5 Breadth first search (BFS) 18](#_Toc521520992)

[3.6 Unknown maze 20](#_Toc521520993)

[3.7 Loading a maze from an image 21](#_Toc521520994)

[3.8 Adding heuristics: A\* search 23](#_Toc521520995)

[3.9 Weighted graphs 24](#_Toc521520996)

[3.10 Challenge 27](#_Toc521520997)

[3.10.1 Building a library of search functions and maze processing functions 27](#_Toc521520998)

[3.10.2 Finding the shortest path between cities 27](#_Toc521520999)

[3.10.3 Around the world in the shortest time 27](#_Toc521521000)

[Chapter 4 Solving puzzles 28](#_Toc521521001)

[4.1 8-Puzzle 28](#_Toc521521002)

[4.2 Service functions 29](#_Toc521521003)

[4.3 BFS 31](#_Toc521521004)

[4.4 A\* 33](#_Toc521521005)

[4.5 Challenge 35](#_Toc521521006)

[Chapter 5 36](#_Toc521521007)

[5.1 Tic-Tac-Toe: The game 36](#_Toc521521008)

[5.1.1 The board 37](#_Toc521521009)

[5.1.2 Players 38](#_Toc521521010)

[5.1.3 The game 39](#_Toc521521011)

[5.2 Tic-Tac-Toe: Minimax approach 40](#_Toc521521012)

[5.3 Tic-Tac-Toe: Q-learning 44](#_Toc521521013)

[5.4 Challenge 51](#_Toc521521014)

[Chapter 6 52](#_Toc521521015)

[6.1 Background and basic ideas 52](#_Toc521521016)

[6.1.1 Single-layer perceptron 53](#_Toc521521017)

[6.1.2 Multi-layer perceptron 57](#_Toc521521018)

[6.2 Neural networks and rules of “Life” 58](#_Toc521521019)

[6.3 Learning the rules of “Life” 61](#_Toc521521020)

[6.4 Challenges 64](#_Toc521521021)

[Appendix A Python basics 65](#_Toc521521022)

[A.1 Programming: Basic concepts 65](#_Toc521521023)

[A.2 A few words on programming environments 65](#_Toc521521024)

[A.3 Basic operations 66](#_Toc521521025)

[A.4 Functions in Python 67](#_Toc521521026)

[A.5 Conditions 68](#_Toc521521027)

[A.6 Loops 68](#_Toc521521028)

[Exercises for A.1-A.6 69](#_Toc521521029)

[A.7 File input/output 70](#_Toc521521030)

[A.7.1 Text files: Reading/writing 70](#_Toc521521031)

[A.7.2 Command line arguments. Modules in Python. 71](#_Toc521521032)

[A.8 File input/output, parsing strings 72](#_Toc521521033)

[Exercises for A.7-A.8 74](#_Toc521521034)

[A.9 Image processing 74](#_Toc521521035)

[Exercises for A.9 75](#_Toc521521036)

[A.10 Python modules 76](#_Toc521521037)

[A.11 Random numbers 77](#_Toc521521038)

[Exercises for A.11 78](#_Toc521521039)

[A.12 Classes 78](#_Toc521521040)

[A.13 Saving Python structures. JSON. 81](#_Toc521521041)

[A.14 Python: Machine learning libraries 82](#_Toc521521042)

[A.15 Coding problems 84](#_Toc521521043)

**Chapter 1**

**Introduction and the course plan**

Recent developments in artificial intelligence and machine learning are quickly propagating into everyday life. It’s hard to find an area of human activity where autonomous agents are not involved in one way or another—from banking and retail to particle accelerators, medicine, and government. Everywhere we have become more and more deeply dependent on automated decision making. This dependency creates a demand among a wide range of professionals for a better understanding of basic principles behind machine learning techniques.

This course is designed to give high school students an introduction to the subject. We’ll present important techniques without going into technical details and provide a perspective for future studies and developments.

The course starts with a traditional approach. First we consider the application of artificial intelligence to finding a path in a maze, and then we discuss navigating graphs and finding short paths.

The next topic is solving puzzles: building decision trees and reviewing ways to improve performance. After that we’ll move to two-player games and learn how to build a rule-based game engine for Tic-Tac-Toe. After reviewing limitations of the approach we’ll consider reinforcement learning—the technique behind recent achievements in chess, Go, and many other applications from finance to automated translation—to automatically build a computer Tic-Tac-Toe player that can compete with a human.

The last part of the course is an introduction to neural networks. We won’t go into details of neural network training, as this would require deeper knowledge of mathematics and optimization concepts. Instead we’ll review the architecture of a perceptron and use available programming modules to reverse-engineer rules of Conway’s Game of Life to give an idea how autonomous agents may derive properties of an unknown environment and operate in it.

The course includes an Appendix on Python programming. Students not familiar with this programming language will find all they need to complete the course.

The course contains three challenges. The work on them can be organized as a competition between groups of students.

# Chapter 2 Simple decisions

## Finding your path

We begin with reviewing how you make everyday life decisions. Let’s say you are on the street and now you want to find your classroom. You can choose between two buildings: Movie Theater and School.



You know that your classroom is in the school. You have two choices, and you enter the school building. The next question you need to answer is which stair to choose: on the left from the entrance or on the right:



Let’s say you go to the right, and this is your second choice. Now you need to choose the floor: second or third:



Your choice (third floor) brings you the next set of options: math classroom, library, or chemistry lab. You choose the math classroom as your destination.



Figure 2.1: Path from street to classroom.

It’s good to know your path in advance, but let’s change the problem: You don’t know the path, but you can identify the classroom as soon as you are there. Now you have more options:



Figure 2.2: Adding choices to the path

In computer science (CS) the structures drawn in Figures 1.1 and 1.2 are called trees, and they play an important role in a number of applications. Tree search is one of the central algorithms in building intelligent agents that operate in known and unknown environments.

The assumption that each decision is made based on a finite set of options (from the street you can enter the school or the movie theater, the right stair brings you to the second or third floor) is a simplification, but a powerful one. At least in principle, this allows you to find a solution by reviewing all possible combinations of decisions. Our first task will be to find a path to your classroom given the tree from Figure 2.2.

## How to navigate a tree

How do we navigate a tree? In other words, how do we go over all nodes starting from the root? The answer depends on the desired order. Let’s consider the following tree:



Figure 2.3

The order of nodes can be 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 or 1, 2, 4, 7, 5, 8, 9, 3, 6, 10—we either go layer by layer or left to right. The first path is called “in breadth,” while the second path is “in depth.” Each path is associated with a tree search algorithm: a “depth first search” (DFS) or a “breadth first search” (BFS).

We’ll start with a DFS and first make an obvious observation: **a part of a tree is a tree**. A part of the tree above is the tree:



This observation gives us an idea of how to run a search. Suppose we want to find a node with a given name X (6 from Figure 2.3, “Math classroom” from Figure 2.2).

* + 1. We start with the root node.
    2. We check the name; if the name is X, we are done.
    3. If it’s not, we are looking into the list of children.
    4. If it’s empty, we are also done, and the search failed: The tree doesn’t have the node X.
    5. If it’s not empty, we have a new tree with each child as a root.
    6. For each child node, we perform a search taking each child node as a root for a new, smaller tree.

Let’s see how this process works for searching for node 6 in the tree from Figure 2.3.

1. Root is 1.
2. It’s not 6, and we continue with two trees: the first with root node 2 and the second with root 3.
3. Node 2 is not 6. We continue with trees starting with nodes 4 and 5.
4. Node 4 is not 6, and we continue with the tree starting with node 7.
5. Node 7 is not 6, and it doesn’t have children.
6. We continue with root 5
7. Node 5 is not 6, and we continue with the trees starting with nodes 8 and 9.
8. Similar to node 7, nodes 8 and 9 are not 6, and they don’t have children.
9. We continue with the tree starting with node 3.
10. Node 3 is not 6, and we continue with its child node, which is 6. **We are done!**

## How to describe a tree

**Python:** You need to learn basic operations and collections in Python: Lists, dictionaries

Let’s see how we can describe the tree from Figure 2. Each node has a name—for example, “School”—and a list of children (“Left stair,” “Right stair”). The list of children can be empty, and in this case the node is called a leaf or a terminal node. The node without a parent is called a root node; in our case it’s “Street.”

Here is how we define the upper part of the tree [[1]](#footnote-1):

#### (Street, (Movies, School))

Adding one more layer:

#### (Street, ((Movies, (Ticket office)), (School, (Left stair, Right stair))))

This approach doesn’t look convenient and transparent; adding one more layer will make the description unreadable. What would help is to describe the tree incrementally, node by node:

Tree1 = (Left stair,[])

Tree2 = (Right stair, [])

Tree3 = (School, [Tree1, Tree2])

Listing 2.1: Python: tree description

|  |
| --- |
| Tree1 = (“Math classroom,” [])  Tree2 = (“Library,” [])  Tree3 = (“Chemistry lab”, [])  Tree4 = (“Library”, [])  Tree5 = (“Restroom”, [])  Tree6 = (“2nd floor”, [ Tree4 , Tree5 ] )  Tree7 = (“ 3rd floor” , [ Tree1 , Tree2 , Tree3 ] )  Tree8 = (“ Right stair” , [ Tree6 , Tree7 ] )  Tree9 = ( “Basement” , [ ] )  Tree10 = ( “Gym” , [ ] )  Tree11 = ( “Left stair” , [ Tree9 , Tree10 ] )  Tree12 = ( “ S c h o o l” , [ Tree8 , Tree11 ] )  Tree13 = ( “Ticket office” , [ ] )  Tree14 = ( “Movies” , [ Tree13 ] )  Tree15 = ( “Street” , [ Tree12 , Tree14 ] )  **print** Tree15 |

Here is what we did: We created Tree1 as a pair of the node name and the list of children (this list is empty, because stairs here are terminal nodes: they don’t have children). The same is true for Tree2. The list for Tree3 is not empty: It’s constructed from two previously defined trees. We reuse trees to avoid nested definitions.

The remaining part of the description: Tree4 = (Ticket office, [])

Tree5 = (Movies, [Tree4]) Tree6 = (Street, [Tree3, Tree5])

Here is the full initialization for the tree from Figure 1.3 in Python:

The next step is to learn how to use this tree representation to navigate a tree.

Our first task is to go through all nodes and print their names.

Let’s say we start from the root node. Our steps are:

* + 1. Print the name of the root node.
    2. If the node doesn’t have children, we are done.
    3. We know we have child nodes, and we remember that each child is a root for its own tree. So we need to do steps 1–3 for all children.

**Assignment:** Use steps 1-3 to manually reproduce the steps for searching node 6 in Figure 1.3.

## Recursion

**Python:** You will need to learn about functions, loops, and conditional statements.

To solve our problem and print tree node names, we need to make one important observation: **A function can call itself**. This is called **recursion** and is of particular importance in computer science, programming, and linguistics. Before we turn to tree navigation, let’s look at the example traditionally used to demonstrate the use of recursion: calculation of factorials. A factorial of a positive integer number *n* is denoted as *n*! and defined as a product of all integers from 1 to n:

*n*! = 1 *·* 2 *· · · · · n*

For example:

3! = 1 *·* 2 *·* 3

5! = 1 *·* 2 *·* 3 *·* 4 *·* 5

Two observations:

1. 1! = 1

2. *n*! = *n ·* (*n −* 1)! For example, 5! = 5 *·* 4!

Let’s first imagine and then program the following function “factorial”:

1. *factorial*(1) returns 1
2. *factorial*(*n*) for *n* ≠1 returns *n·factorial(n-1)*

Here is the code for factorial calculation:

|  |
| --- |
| **def** factorial(n):  **if** n == 1 : **return** 1  **return** n \*factorial(n*−*1)  **print** factorial(5) |

*F* (1) = 1

*F* (2) = 1

*F* (*i*) = *F* (*i −* 1) + *F* (*i −* 2)

The first 6 numbers are: 1, 1, 2, 3, 5, 8. Write two versions of the function: using loops and using recursion.

* Write function that print the first N Fibonacci numbers. Fibonacci numbers *F* (*i*) are given by the following formulas:
* Write function that calculates factorials, don’t use recursion.

recursion.

* Add printing function parameter to see the order of function calls in the

**Assignments:**

## Navigating a tree (depth first)

Now we can solve the problem of printing node names. Here is the Python code: It’s the implementation of items 1–3 above.

Listing 2.3: Python: printing node names (depth first)

|  |
| --- |
| **def** print\_node\_names(T):  print T[0]  **for** child **in** T[1]: print\_node\_names(child) |

Let’s look at these three lines of code in more detail.

* + 1. Here we are defining the function of one parameter. The parameter is a tree.
    2. The tree is represented by a pair, and the first element T[0] of the pair is the name. We print it.
    3. The second element T[1] of the pair is the list of the node’s children. Line 3 traverses all children and calls print node names for each of them.

Now we apply this function to print all nodes of the tree we introduced above.

**Assignments:**

* Call the function print node names for the tree defined above.
* Change the function print node names to print nodes in the reverse order.

## Additional Exercises

For trees from the next page:

1. Find all root nodes
2. Find all terminal nodes
3. Manually navigate trees (depth first) – write the sequence of nodes
4. Create tree description and confirm 3) with your **print\_node\_names** function
5. Current version of **print\_node\_names** function navigates nodes left-to-right. Change the function to navigate right-left. For example for the following tree:



print “2 5 9 8 4 7” instead of “2 4 7 5 8 9”.

a) b) c)



d) e)



f)

# Chapter 3 Labyrinths

## A maze as a tree

Let’s apply tree navigation techniques to an old problem: finding a path through a maze. To begin with, we’ll assume that the maze is known (we have a full description of the maze before we start). Here is a simple maze:



Our goal is to find a path through the maze from **A** to **B**:



The first question we need to answer is about the maze description. The answer comes from the following observation: Every time we make a decision about navigating the maze we make it from a finite set of choices:



When we are in location **7** we can move to location **8**; when we are in **8** we can move to

**5** or **9**. Let’s enumerate all locations and list all possible moves:

*A →* 1*,* 1 *→* 4*,* 4 *→* 7*,* 7 *→* 8*,* 8 *→* 5 *or* 9*,* 5 *→* 2*,* 2 *→* 3*,* 3 *→* 6 *or B.*

We know this structure from the previous chapter. It’s a tree:



**Assignments:** Describe this tree in Python and print the path from A to B.

## Maze descriptions

**Python:** You need to learn file I/O, string processing, and list comprehension in Python.

The maze from the figure above is small, and it’s easy to describe it by manually tracing all possible moves. For bigger mazes we need an automated way of extracting trees from schematic representations. For the maze above, we can use the following:

0100000

0101111

0101010

0101000

0111110

0000000

Zeros represent walls, and ones represent paths. You may find that the size—the number of elements—is different. This is the cost of introducing walls: They have a size in the scheme. Extracting a maze from an image file (scanned or drawn) is technically possible, we’ll considerate later in the course.

You can define the scheme of a maze inside your Python program, but a more flexible approach is to save schemas as text files and have one program that can work with different mazes. The following piece of code combines reading the file, splitting it into lines, and processing lines element by element:

Listing 3.1: Python: Reading a maze description

|  |
| --- |
| **import** sys  f = **open**(sys.argv[1],”r”)  maze = [[x == “1“ **for** x **in** line] \  **for** line **in** f.read().split(“\n“) **if len**(line)>0]  f.close()  **print** maze |

## Converting a maze to a tree. Graphs.

Now we have a geometrical description of a maze: For each element we can tell if it’s a path or a wall. Let’s now convert it into a tree representing a node and its children, or in the case of a maze, an element of a path and elements of a path on the right-left-top-bottom (if they exist).

To make the work with maze descriptions more convenient we need to change the way we identify the elements of a maze. Earlier we used simple enumeration. For the small 3x3 maze above we enumerated all 9 elements from 1 to 9. For a bigger maze, enumeration doesn’t give a clear picture. Where, for example, is the element 134 in the 18x13 maze? Instead of enumeration we’ll use two numbers: index of a line and index of an element’s position in a line. Element 1 in this notation becomes (0,0), element 8, (2,1). Note: Python uses zero-based indices for its lists. Enumeration can be easily restored if needed: For a position (*i, j*), the “old” index is *N\*i*+*j*+*1*, where *N* is the width of a maze: 1 = 3\*0+0+1*,* 8 = 3\*2+1+1.

*∗*

* *∗*

The use of enumeration or index pairs makes tree definition a little simpler. Each index is unique, and we don’t need to preserve a path to a node to distinguish nodes with identical names. A tree can be defined using a dictionary mapping a node’s unique name to a list of children.

The next step is to examine the maze and create the tree. We’ll examine the maze and for each path element analyze surrounding elements: For element (*i, j*) we’ll need to check elements (*i-1, j*)*,* (*i*+*1, j*)*,* (*i, j-1*)*,* (*i, j+1*). Adding or subtracting 1 from an index may give a new index outside the maze. We’ll skip such cases and first write a function that returns True if an index is inside the maze, and False if it’s outside:

Listing 3.2: Python: Checking index position

|  |
| --- |
| **def** is\_inside(index,maze\_width,maze\_hight):  # checking first index component  **if** index[0] < 0 **or** index[0] >= maze\_hight: **return** False  # checking second index component  **if** index[1] < 0 **or** index[1] >= maze\_width: **return** False  # in all other cases index is inside  **return** True |

Now we can code the loop over all elements:

Listing 3.3: Python: Creating a tree from a maze

|  |
| --- |
| **def** create\_tree(maze):  # height of the maze is the number of lines in the definition  height = len(maze)  # we are assuming all lines in the definition are of the same length  width = len(maze[0])  # we are creating empty dictionary ...  tree = {}  # ... and first populate it with path elements mapped to empty lists  **for** i **in** range (height):  **for** j **in** range (width):  **if** maze[i][j]: tree[(i,j)] = []  # now for all elements of the tree we populated  # lists with indices of surrounding path elements:  **for** e **in** tree:  i,j = e[0],e[1]  **if** is\_inside((i-1,j),width ,height):  **if** maze[i-1][j]: tree[e].append((i-1,j))  **if** is\_inside((i+1,j),width ,height):  **if** maze[i+1][j]: tree[e].append((i+1,j))  **if** is\_inside((i,j-1),width ,height):  **if** maze[i][j-1]: tree[e].append((i,j-1))  **if** is\_inside((i,j+1),width ,height):  **if** maze[i][j+1]: tree[e].append((i,j+1))  **return** tree |

**Assignment:** Combine reading a maze from a file, maze parsing, index verification, and tree creation into one program. Run the program and review the results.

As we can see, the tree constructed above is different from trees we’ve seen before. Note that a node contains its parent in the list of children: We do not distinguish parents and children, and operate only with neighbors. This creates the following problem: The tree navigation algorithm we used before includes the loop over all children. If a parent node is in the list of children, the loop becomes infinite, or, taking into account that we used recursion, recursion becomes infinite.

A similar problem arises when a maze has loops. As a result, we may start going in circles:



The node structure doesn’t form a tree. The structure where nodes can form loops is called a **graph**. A tree is a graph without loops.

To navigate a graph and find a path between two nodes, we have to modify the algorithm by keeping a record of visited nodes. This is an updated version of a Depth First Search:

Listing 3.4: Python: DFS with saving visited nodes

|  |
| --- |
| **def** DFS(graph,node,visited):  # adding new node to the list of visited  visited.append(node)  # for each node ...  **for** child **in** graph[node]:  # ... if it’s not visitied  **if** child **not in** visited: DFS(graph,child,visited) |

The function now takes two more parameters: current node and the list of visited nodes. When calling the function, we use an entrance node and an empty list.

**Assignment:** Combine tree created from a maze definition with DFS function and print all visited nodes. Use (0,1) as the entrance node.

## Finding the exit

We may see that a **DFS** function doesn’t actually do a search: It traverses **all** nodes. That’s not what we need—the search must stop when we come to the exit. We have to add a new parameter, **goal**,that identifies the exit node, and check each node:

Listing 3.5: Python: DFS with checking the goal

|  |
| --- |
| **def** DFS(graph,node,visited,goal):  visited.append(node)  # stop if we've reached the goal  **if** node == goal: **return** True  **for** child **in** graph[node]:  **if** child **not in** visited:  # mini assignment: why do we need this check?  **if** DFS(graph, child, visited, goal): **return** True  # we'll come here if the maze doesn't have a solution  **return** False |

**Assignment:** Modify the code of the previous assignment and find the path from entrance to exit.

## Breadth first search (BFS)

When we started talking about navigating a tree, we mentioned two algorithms. The first, a depth first search, we’ve learned about in previous sections. Let’s consider the second one, the breadth first search, where we go layer by layer through a tree. For a general graph, BFS first examines all neighbors (children) of a node before increasing the depth. Similar to DFS, BFS keeps track of visited nodes to avoid going in a loop.

The algorithm includes the following steps:

* + 1. For each node we need to maintain a path from the start.
    2. We create a list **Q**,which, for each node keeps a queue of pairs (path,node). In the beginning the list contains only the starting node.
    3. We create an empty list **V** to keep visited nodes.
    4. The main loop continues while we have nodes to visit—or, in other words, while the queue **Q** is not empty.
    5. At each step we take the first element of **Q** as our current node and remove it from the queue.
    6. We stop if the current node is our goal.
    7. We skip the current node if it’s in V.
    8. In all other cases, we append the current node to V and the path and all neighbors of the current node to Q.
    9. We continue the main loop.
    10. We stop if the current node is our goal.
    11. We skip the current node if it’s in V.
    12. In all other cases, we append the current node to V and the path and all neighbors of the current node to Q.
    13. We continue the main loop.

Here is the Python code [[2]](#footnote-2):

Listing 3.6: Python: Breadth first search

|  |
| --- |
| **def** BFS(graph, start, goal):  # initializing queue  Q = [([start], start)]  # initalaizing list of visited nodes  V = [ ]  # main loop while Q is not empty  **while** len(Q)>0:  path, current\_node = Q[0]  # remove current node from the queue  Q = Q[1:]  # are we done ?  **if** current\_node == goal: **return** (path,True)  # have we already been here?  **if** current\_node **in** V: **continue**  # saving current node  V.append(current\_node)  # loop over all neighbors  **for** neighbor **in** graph[current\_node]:  Q.append((path+[neighbor],neighbor))  # we cannot reach the goal: returning empty path  **return** ([],False) |

**Assignments:**

1. Compare DFS and BFS for the following maze:



2. Compare DFS and BFS for a bigger maze:

0 1 0 0 0 0 0 0 0 0 0 0

0 1 1 1 1 1 1 1 1 1 1 0

0 1 0 0 0 0 0 0 1 0 0 0

0 1 0 1 1 1 1 1 1 1 1 0

0 1 0 1 0 0 0 0 0 0 1 0

0 1 0 1 0 1 1 1 1 1 1 0

0 1 1 1 0 0 1 0 0 0 1 0

0 1 0 1 1 1 1 0 1 0 0 0

0 1 0 0 1 0 1 1 1 1 1 0

0 1 0 1 1 0 0 0 0 0 1 0

0 1 1 1 0 0 1 1 1 1 1 1

0 0 0 0 0 0 0 0 0 0 0 0

Create a file with a description of the maze, parse it into a graph, and do the search.

## Unknown maze

In previous sections we assumed that we had a maze description before we started searching. Our approach was to create a full graph and use BFS or DFS to find the exit. What if we don’t have a full maze description, and all we can do is to analyze the environment and distinguish walls and paths? In that case, we can enter a maze, look around, and list all possible moves. We can count our steps to use (i,j) indices for elements of the maze.

For the simple maze we considered in the beginning, we don’t know anything about elements with question marks. We know our current position (0,0), and we see that we can move only to (1,0).



When we are in (2,1), we remember our path and know that we can move to (2,2) and (1,1):



Would it be possible for us to find the path? The answer is yes. We can find the path even if we don’t have a full graph to begin with. We can construct the graph when navigating the maze, accumulating knowledge about the operating environment. If we look at DFS or BFS we can easily find that we don’t need information about the graph, except for the list of already-visited nodes and the list of neighbors.

Our approach will be to start with an empty graph, query the environment (look around), find neighbors, and update the graph with new information.

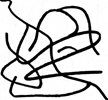
To query the environment we’ll need a function that returns all neighbors for a given (i,j) node. We’ve already done this in the function of converting a maze description into a graph, except that that function traversed all maze elements, and now we need a smaller function for a single element.

Assignments:

1. Write functions that return neighbors.
2. Adjust BFS and DFS functions to navigate a maze with an unknown description in the beginning graph.

## Loading a maze from an image

So far, we have been using small mazes. These mazes have given us some idea of how search algorithms work, but they are not large enough to compare performance and resulting paths. It’s inconvenient to describe a big maze manually by entering 1s and 0s, so let’s load mazes from images. Here is an example of a hand-drawn maze:



You can use a drawing program (PaintBrush or an equivalent) or draw a maze on paper and then scan or make a picture of it with a phone. Make sure the lines are thick enough to avoid problems with image resizing.

You can see an image as a list of pixels going row by row from the top to the bottom. For a tiny image of four rows and three columns, the first three elements of the list (list indices are 0,1,2) describe the first row, elements with indices 3,4,5 describe the second row, etc....

There is a variety of image formats and color modes. Color mode defines the way to describe a pixel’s color. The most popular mode is RGB (red–green–blue), which defines the color as a combination of three numbers representing the red, green, and blue components of a color. Each number is between 0 and 255. For example:

(0,0,0) black

(255,0,0) red

(0,255,0) green

(0,0,255) blue

(255,255,255) white

When all RGB components are equal, the color is grey: (128,128,128), (35,35,35), (233,233,233) are all shades of grey. For greyscale images the color may be represented by a single number between 0 and 255.

We will not address the wider area of image processing. For purposes of the course we need to be able to read the image into a list of pixels, identify the image height and width, identify dark pixels (we’ll consider them as elements of the path), change a pixel’s color to indicate a found path or visited element, and save images into a file.

**Python:** Please refer to Appendix A9 to learn basics of image processing.

**Assignments:**

1. Load a maze from an image file, create a graph, and find paths using DFS and BFS.
2. Load a maze from an image file, and find paths and visited nodes using both DFS and BFS*.*
3. Adjust the image file: Change visited nodes to green and path nodes to red. Compare results.

*a*Depending on your system settings you may get an error message when doing a recursive search with DFS; the number of recursion calls is limited to avoid infinite recursion. If you see this problem, change the recursion limit before calling the DFS function: Use **sys.setrecursionlimit(N)** in Python. Experiment with N: Try different values.

Note on the assignment: Depending on your system settings you may get an error message when doing a recursive search with DFS; the number of recursion calls is limited to avoid infinite recursion. If you see this problem, change the recursion limit before calling the DFS function: Use **sys.setrecursionlimit(N)** in Python. Experiment with N: Try different values.

## Adding heuristics: A\* search

Both DFS and BFS can find a path through a maze if it exists. At the same time, their performance (length of the resulting path and number of visited nodes) may not be satisfactory. Sometimes we have to visit all nodes to find the path. Time and memory requirements can make the search problematic. Can we improve the search? Given the graph of a maze (or in a more general case, the graph of our decisions), we may try to change the way we choose the next step. In previous sections we didn’t distinguish between node neighbors: The order in which we visited them was arbitrary, and didn’t reflect any idea about a desired order.

Can we change the way we choose the next node to improve performance (make a path shorter and reduce the number of visited nodes)? The answer is positive, although the improvement is not guaranteed. We can introduce a new algorithm that will work better **on average**, without improving the so-called **worst case**. In other words, it’s possible to find a case when the algorithm doesn’t improve performance, but most of the time it works better.

What would be an “improved” way of choosing the next node? We are somewhere inside a maze and have several choices. Intuitively we can choose a node that is closer to the exit than others: If we know the exit is on the right, we may want to move to the right neighbor node if it’s available. More accurately, we need to choose the neighbor node to minimize the number of steps from the beginning to the node plus our estimation of the distance from the node to the end. The method is called A\* (A-Star). Let’s try to formalize it.

First, we’ll need a function that returns our estimate of the distance from a node to the goal. Given that we can move only up-down-left-right, the distance between two nodes (*i*1*, j*1) and (*i*2*, j*2) can be expressed as a sum of distances by each direction:

*|i*1 *− i*2*|* + *|j*1 *− j*2*|*

or as a Python function:

Listing 3.7: Distance between nodes

|  |
| --- |
| **def** distance(node1,node2):  **return abs**(node1[0]-node2[0]) + **abs**(node1[1]*−* node2[1]) |

Similar to BFS, we’ll keep the queue for nodes to visit, but in A\* those nodes have to be sorted based on the measure we discussed above (the number of steps from the beginning plus the expected distance). This functionality can be implemented with basic Python lists, but it’s more convenient to use already-available modules. Python provides all we need with the module **heapq**.

Function **heappush** adds a new element to the sorted queue, preserving the order. For example, adding 2 to the list of [1, 3, 4, 5] creates a new list on [1, 2, 3, 4, 5]. If an element has several components **heappush**,the sort order is according to the first component.

Function **heappop** returns the first (“smallest”) element from the queue and removes it from the queue.

The queue element for an A\* search must contain the following components:

* + 1. Number of steps from the beginning plus an estimate of the distance to the end:

the value we’ll use to sort the elements (remember, we want to prioritize nodes with smaller values).

* + 1. Number of steps from the beginning.
    2. Path from the beginning.
    3. Node.

Listing 3.8: Python: A\* search

|  |
| --- |
| **from** heapq **import** heappop  **from** heapq **import** heappush  **def** AStar(graph, start, goal):  Q = []  heappush(Q,(distance(start,goal),0,[start],start))  V = []  **while len**(Q) *>*0:  dist, from\_start, path, current\_node = heappop(Q)  **if** current\_node == goal: **return** (path,V,True)  **if** current\_node **in** V: **continue**  V.append(current\_node)  **for** neighbor **in** graph[current\_node]:  dist = distance(neighbor, goal)  heappush(Q,(from\_start+dist, from\_start+1,\  path+[neighbor], neighbor))  **return** ([],[],False ) |

You may find that the code is pretty similar to the code for BFS. The differences are in the components of **Q** discussed before, and in the use of **heapq** instead of plain Python lists.

**Assignments:**

1. Load a maze from an image file, create a graph, and find paths using A\*.
2. Adjust the image file: Change visited nodes to green and path nodes to red. Compare results (paths and visited nodes) with DFS and BFS.
3. Experiment with the distance function: Try to introduce the distance between nodes in a different way. Compare results.

## Weighted graphs

As we already know, a graph is defined by nodes and edges. One can conceptualize nodes as dots and edges as lines connecting a pair of nodes. A tree is a special case of graphs: a graph without loops. We used different ways to name nodes: words, when describing the path from street to classroom, or (i,j), for a pair representing a node’s position in a maze. Going forward, we’ll enumerate nodes or assign them letters for compactness.

So far, all edges have been identical: No weight has been assigned to an edge. At the same time, in some applications weight can be introduced in a very natural way. If, for example, nodes are cities connected by roads, we can think about weights as distances between cities. Graphs with weights assigned to edges are called weighted graphs.[[3]](#footnote-3) They appear in a number of problems of logistics and planning when the cost of each decision is different. Consider the famous “travelling salesman problem”: A salesman has to visit several cities and knows the distances between them (not all cities are connected by roads). He must choose the shortest path to visit all of them. In this case, cities are nodes, and edge weights are distances between cities connected by roads.

Consider the following graph:

11

B

D

3

8

2

4

1

3

7

A

C

What is the shortest path from A to D? Let’s review possible paths:

*A → D* : 8

*A → B → D* : 3 + 11 = 14

*A → C → D* : 7 + 2 = 9

*A → B → C → D* : 3 + 1 + 2 = 6

The shortest path directly from A to D appears not to be the shortest one that takes into account edge weights. You can easily imagine how this happens in reality: The shortest path may be the steepest one, or part of the path may include toll roads, etc. The choice of the shortest path must take weights into account, and depending on the specific problem weights may reflect actual distances, tolls, traffic, flight times, or road conditions.

By this time, you already know several path-finding algorithms. Adding weights will not change much.

**Assignment:** Think about which algorithm out of those you already know—DFS, BFS, A\*—may be used for finding paths in weighted graphs. Hint: The algorithm must explicitly account for the path length.

To implement the search in a weighted graph, we first must learn how to describe such graphs: how to add weights to lists of neighbors. A simple way of doing this in Python is to use dictionaries and describe each neighbor not only by name, but also by weight. The graph from the picture above:

Listing 3.9: Python: Weighted graphs

|  |
| --- |
| graph = {}  graph["A"] = [("B", 3), ("C", 7), ("D", 8)]  graph["B"] = [("A", 4), ("C", 1), ("D", 11)]  graph["C"] = [("D", 2)]  graph["D"] = [("C", 3)] |

Here each neighbor is given by the pair of the name and weight.

One of the best-known algorithms for finding short paths in weighted graphs is Dijkstra’s algorithm. It can be viewed as an A\* that takes into account only the path from the beginning and doesn’t take into account any estimates for the distance to the end.

With that said, changes will affect just a couple of lines of the original A\* implementation. We’ll use **heapq** to maintain the queue according to the distance from the start:

Listing 3.10: Python: Dijkstra search

|  |
| --- |
| **from** heapq **import** heappop  **from** heapq **import** heappush  **def** Dijkstra( graph, start, goal) :  Q = []  heappush(Q,(0,[start],start))  V = [ ]  **while** len(Q)>0:  from\_start, path, current\_node = heappop (Q)  **if** current\_node == goal: **return** (path,V,True)  **if** current\_node **in** V: continue  V.append(current\_node)  **for** neighbor, weight **in** graph[current\_node]:  heappush(Q,(from\_start+weigth,\  path+[neighbor], neighbor ) )  **return** ([],[],False) |

**Assignment:** Combine weighted graph definition and the code of Dijkstra’s algorithm, and run the program. Adjust the code to output the distance from the beginning and the number of steps along the path.

## Challenge

### Building a library of search functions and maze processing functions

The goal is to learn how to write reusable code.

**Python:** Please refer to the Appendix to learn module programming: file organization and module function access.

By this time, you have already developed several useful functions:

* + - 1. Search functions: DFS, BFS, A\*, Dijkstra.
      2. Maze operations: loading from a maze description, loading from an image file.
      3. Creating a graph from a maze description.

The first challenge in this course is to create a Python module providing this functionality. Keep search and maze-related functions separate.

After you are done writing the module, review your assignments and complete them using the module you created.

### Finding the shortest path between cities

Please refer to the road map of your country, select 10 to 15 cities, and find the mileages between them. Create a graph that represents your findings. Use Dijkstra’s algorithm to find the shortest paths between distant cities.

Use your knowledge of A\* and Dijkstra’s algorithms and come up with a heuristic function to improve the search performance: add an estimate of the distance from the current node to the goal using Dijkstra’s algorithm.

### Around the world in the shortest time

Please refer to an international flight schedule, find 10 to 20 cities around the world, and identify flights between them. Collect flight and connection times, create a graph description, and find the shortest time to travel around the world. Think about how you would deal with multiple flights from/to one city.

# Chapter 4 Solving puzzles

The problems we have been solving so far had a static environment, in which problems didn’t change after our decisions: Mazes didn’t change after our moves, graphs didn’t change when we were searching nodes. In contrast, real-life environments where an autonomous agent is operating may react and change as a result of the agent’s decisions. Games like chess, checkers, and Go give good models of changes after players’ moves.

## 8-Puzzle

In this section we’ll consider 8-Puzzle. The puzzle consists of a 3x3 board and 8 tiles numbered from 1 to 8. In the beginning all tiles are randomly placed on the board:



Tiles can slide to the empty space. In the example above, a player can move tiles 2, 3, 4, or 6. The goal is to bring tiles into numerical order:



It’s important to know that half of all randomly generated initial tile positions correspond to non-solvable puzzles. Any configuration where only two tiles are swapped is not solvable (we’ll leave this statement without a proof). For example:



(swapped 7 and 8) is not solvable.

Before we start solving the puzzle we need to learn how to describe it. As before, with a maze, we’ll use a pair of indices (i,j) to refer to a specific position:



The first index in a pair identifies a row, the second index, a column. This representation is convenient for finding neighbors. For example, neighbors of (1,0) are (0,0), (1,1), and (2,0): One element of a pair differs from (1,0) by one. Note that we need to check boundaries: An index cannot be less than 0 or more than 2.

For other purposes this representation is less convenient, and a list or a tuple of 9 numbers may be preferable (soon we’ll see how to use two representations together). For example:

(1*,* 3*,* 2*,* 5*,* 7*,* 4*,* 0*,* 8*,* 6)

corresponds to



Note that we are using 0 to describe the empty pace. To begin, we need several service functions:

* + 1. We need a function that returns position indices for any number on the board. For example, for the board above we need a function that returns (1,2) for 4.
    2. We need a function that returns all possible moves for a given board. For example, for the board above we need a function that returns 5 and 8.
    3. We need a function that returns an updated board after a move. For example, moving 5 (sliding tile 5 down to the empty space) for the board above produces a new board (1*,* 3*,* 2*,* 0*,* 7*,* 4*,* 5*,* 8*,* 6)
    4. We need to be able to compare two boards and return True for identical boards: In other words, we need to check if the current board is our goal (1*,* 2*,* 3*,* 4*,* 5*,* 6*,* 7*,* 8*,* 0)
    5. We need to be able to generate a random but solvable board.

**Python:** You need to learn how to generate random numbers in Python to continue.

## Service functions

1. Position of a particular tile **n** on a **board**. The board is given as a list of 9 numbers:

Listing 4.1: Python: 8-puzzle, position of a tile

|  |
| --- |
| **def** position(board,n):  # go over all rows ...  **for** i **in** range(3) :  # ... and over all columns  **for** j **in** range(3) :  **if** board[3\*i+j] == n: **return** (i,j) |

We just go row by row and column by column and return a pair of indices for the required tile.

1. All moves:

Listing 4.2: Python: 8-puzzle, all moves

|  |
| --- |
| **def** moves(board):  # position of an empty cell  i, j = position(board,0)  all\_moves = []  # above the empty cell  **if** i-1>=0: all\_moves.append(board[(i-1)\*3+j])  # below the empty cell  **if** i+1<=2: all\_moves.append(board[(i+1)\*3+j])  # to the left of the empty cell  **if** j-1>=0: all\_moves.append(board[i\*3+j-1])  # to the right of the empty cell  **if** j+1<=2: all\_moves.append(board[i\*3+j+1])  **return** all\_moves |

Here we create a list of all neighbors of the empty cell: above, below, to the left, and to the right.

1. Board after a move. The move is given by a tile number.

Listing 4.3: Python: 8-puzzle, making a move, returning a new board

|  |
| --- |
| **from** copy **import** copy  **def** move(board,n) :  # initializing new board as a copy of the board  new\_board = copy(board)  # position of the empty cell  i0, j0 = position(board,0)  # position of the moving cell  i,j = position(board,n)  # swap: a,b=b,a  new\_board[i0\*3+j0], new board[i\*3+j] = \  new\_board[i\*3+j], new\_board[i0\*3+j0]  **return** new board |

Note the new element: In the first line we import the function **copy** from the module **copy**. We need the **copy** function to create a copy of the original board element by element.

1. Comparing two boards

Listing 4.4: Python: 8-puzzle, comparing board

|  |
| --- |
| **def** cmp\_boards(b1,b2):  **return tuple**(b1) == **tuple**(b2) |

Note that we need to compare two boards given by lists component by component. This can be done either in a loop, or by converting lists into tuples. In Python, tuples are compared by components.

1. Generating a random solvable board.

Listing 4.5: Python: 8-puzzle, generating a board

|  |
| --- |
| **from** copy **import** copy  **from** random **import** random  **from** random **import** randint  **def** generate(board):  new board = copy(board)  **for** \_ **in** range(randint(10,20)) :  all\_moves = moves(new\_board)  random\_move\_index = randint(0,len(all\_moves)-1)  random\_move = all\_moves[random\_move\_index]  new board = move(new\_board,random\_move)  **return** new\_board |

We start with some board and then make several randomly chosen moves. Here the number of moves is also random: between 10 and 20. In the loop we first find all possible moves for the current board. Next, we choose a random move from the list of all possible moves and make the move, updating the board. If we start with a solvable board ([1,2,3,4,5,6,7,8,0]) we’ll generate a solvable initial board.

**Assignments:**

1. Write a function that prints a board. Use space instead of 0 for the empty space. For example, the board given as a list [1,3,7,0,5,6,8,2,4] is printed as

137

56

824

1. Write a function that generates and prints a solvable board.

## BFS

Now, as we have all service functions, let’s implement BFS for solving the puzzle. The code is pretty similar to the BFS we developed for finding a path in a maze (Listing 4.6.).

Similar to a maze problem, we are creating a queue and a list of boards we have already tried. The queue element is a pair: a list of all moves made before and the resulting boards. While we have anything in the queue we take its first element. If the board is our goal, we return a True completion flag along with the list of moves we made to solve the puzzle. If not, and the board (**cur**) is in the list of already visited moves, we skip it. Otherwise, we continue with the board, beginning by appending it to the list of visited moves. The next step is similar to the loop in the BFS for mazes, except that neighbors are substituted with all possible moves. In this step, we append pairs of moves and resulting boards to the queue.

Listing 4.6: Python: 8-puzzle, BFS

|  |
| --- |
| **def** Puzzle8BFS(board, goal):  Q = [([],board)]  V = []  **while** len(Q)>0:  path, cur = Q[0]  Q = Q[1:]  **if** cmp\_boards(cur,goal): **return** ( True , path )  **if** tuple(Q) in V: **continue**  V.append(tuple(cur))  **for** m **in** moves(cur) :  Q.append((path+[m],move(cur,m)))  **return** (False, []) |

You may find that the time needed to solve the puzzle can be significant. Let’s add some measures to quantify time and memory requirements: the number of steps needed to solve and the maximum size of the queue:

Listing 4.7: Python: 8-puzzle, BFS with counters

|  |
| --- |
| **def** Puzzle8BFS(board, goal):  Q = [([],board)]  V = []  counter = 0  max\_size = 0  **while** len(Q)>0:  path, cur = Q[0]  Q = Q[1:]  **if** cmp\_boards(cur,goal):  **return** ( True, counter, max\_size, path )  **if** tuple(Q) in V: **continue**  V.append(tuple(cur))  **for** m **in** moves(cur) :  Q.append((path+[m],move(cur,m)))  **if** max\_size < len(Q): max\_size = len(Q)  counter += 1  **return** (False, counter, max\_size, []) |

**Assignment:** Combine all service functions with a BFS function, generate an initial random board, and solve the puzzle. Experiment with the number of random moves used to generate initial board. Change (10,20) to (100,200) or any other bigger numbers.

Now we can see how different the numbers of steps and the queue sizes for different initial boards are.

## A\*

The number of moves and the resulting execution time may be not satisfactory, so we may need to improve the algorithm.

Similar to finding paths in mazes, we need to rank possible moves: We need to order them based on some estimate of a move’s quality. For mazes the estimate was based on the geometrical distance from the current node to the goal. Here we need to introduce a distance from a current board to the goal.

Let’s start with a simpler question: how to introduce the distance if we have only one tile and 8 empty spaces. The tile is moved out of its expected place:



1 has to be in the top left position. We need to make two moves up and one move to the left. We can consider this number of moves from the current position of a tile to the goal as the distance for a single tile. The sum of these distances for all tiles is the distance for the board. Consider the following board:



We need:

1. moves for the tile 1
   1. moves for the tile 2
2. move for the tile 3
3. moves for the tile 4

2 moves for the tile 5

2 moves for the tile 6

1. moves for the tile 7
2. moves for the tile 8

Total distance is 0 + 2 + 1 + 2 + 2 + 2 + 2 + 3 = 14

Here is the code that implements the distance calculations:

Listing 4.8: Python: 8-puzzle, distance for boards

|  |
| --- |
| **def** dist(b1,b2):  total\_dist = 0  **for** i1 **in** range(3):  **for** j1 **in** range(3):  i2, j2 = position(b2,b1[i1\*3+j1])  total\_dist += abs(i1-i2) + abs(j1-j2)  **return** total\_dist |

We go in the loops over 3 rows and 3 columns. For each cell (*i*1*, j*1) we have the tile number on the first board and can get the position (*i*2*, j*2) of that number on the second board. Next, we increase the total distance by . Note that the procedure also accounts for the empty cell.

Now, as we have sketched out the distance from the current board to the goal, the next step should already be clear: We can write a version of an A\* search algorithm.

Listing 4.9: Python: 8-puzzle, A\*

|  |
| --- |
| **from** heapq **import** heappop  **from** heapq **import** heappush  **def** Puzzle8AStar(board,goal):  Q = []  heappush(Q,( dist(board, goal),[],board))  V = [ ]  counter = 0  max\_size = 0  **while** len(Q) > 0:  \_, path, cur = heappop(Q)  **if** cmp\_boards(cur, goal):  **return** (True, counter, max\_size, path)  **if** tuple(Q) in V: continue  V.append(tuple(cur))  **for** m **in** moves(cur):  m\_board = move(cur,m)  heappush(Q,(dist(m\_board,goal),path+[m],m\_board))  **if** max\_size < len(Q): max\_size = len(Q)  counter += 1  **return** (False, counter, max\_size, []) |

As before, we use the **heapq** module to work with sorted elements. Each element of the queue has three components now: the distance to the goal (it’s the first element used for sorting), the list of moves executed from the original board, and the board.

**Assignment:** Combine all functions we’ve developed in this chapter, generate initial boards, and solve the puzzle with BFS and A\*. Compare the numbers of steps and maximum queue sizes in BFS and A\*. Compare the number of moves needed to solve the puzzle with BFS and A\*.

You may find that A\* works faster and requires less memory, but it may produce the puzzle solution with more moves than BFS. A\* doesn’t guarantee the smallest number of moves, but on average it works faster than BFS.

## Challenge

* + 1. Implement the N-puzzle: Adjust the code developed in this chapter to make the size of the board a parameter. For example, take a 15-puzzle (4x4 board)
    2. Implement this puzzle on a non-square board, for example on a board that is 4x3
    3. Try to introduce a different distance between the board and the goal. In this chapter we’ve introduced the distance as a sum of the distances of all tiles. You may consider, for example, not a sum, but a maximum distance. You can generate an initial board and then compete for the smallest number of steps of the A\* algorithm.

# Chapter 5

**Multi-player games**

In this chapter we’ll consider decision-making in multi-player games: the case when the state of a game can change unpredictably due to an opponent’s move. Games like chess, checkers, and Go are all good examples of two-player games. Building a computer program that can play these games is difficult, but the principles behind several approaches can be studied on a game as simple as Tic-Tac-Toe.

## Tic-Tac-Toe: The game

Two players make moves on a 3x3 board. They start with an empty board

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |

The first player puts an X in a free space, and the second player puts an O. The player who first puts 3 symbols in a row is the winner. Example of winning position:



There may be conditions when the board is full and there is no winner (draw):



We’ll describe the position on the board by the list of 9 symbols: “X”, “O”, or “ ” (empty space). For example, the two positions above will be [“X”, “O”, “O”, “ ”, “X”, “ ”, “ ”, “ ”, “X”], and [“X”, “O”, “X”, “O”, “X”, “O”, “O”, “X”, “X”].

**Python:** To continue you need to learn elements of object-oriented programming: classes in Python, constructors, member functions, and variables.

There are several approaches to building a computer program that can play Tic- Tac-Toe or other similar two-player games:

* + 1. We can hard-code some strategy: a set of rules the program applies to choose the next step. For example, two obvious rules are: “**Win**: If you have two symbols in a row and can put in the third symbol, do so and win”; or “**Don’t let the other player win**: If on the next step your competitor can win by putting in the third symbol, prevent this.”
    2. The first approach is hard to generalize because it’s close to impossible to apply it to a game just a little more complex than Tic-Tac-Toe. We can introduce a general way of choosing the next move that increases the chances of a win: the so called minimax approach.
    3. We’ll find that minimax may require a lot of resources to evaluate a move and that the next approach will be the reinforcement learning (Q-learning) to construct an evaluation function. The program will first play against itself for some time and will eventually build a model that can play against a human or another program.

Let’s start with a simple rule-based strategy: We’ll implement the simplest case when the program makes a random move. We’ll leave the development of a stronger rule-based strategy for the challenge at the end of the chapter.

We have to prepare several service elements:

1. The class that implements the board and board-related functions.
2. Two classes that implement a computer player and a human player. The computer player makes a move based on one or the other algorithm, and the human player enters moves from the console.
3. The class that implements the game: the order of operations, position evaluation, etc.

All those entities—the board, the players, and the game—have their own data and functionality, so it’s convenient to organize these entities as class member variables and functions.

### The board

Let’s first review the functionality related to the board and not related to the players’ decisions, the order of the players’ moves, etc. Any implementation of the board has to be able to do the following:

* + - 1. Initialize the empty board.
      2. Check if the board is filled: no moves available, or there is a winner.
      3. Check if a player is a winner.
      4. Change its own state on a move.
      5. Find all possible moves (indices of all empty cells).
      6. Return human-readable representations.

Here is the implementation:

**\_\_init** initializes a buffer of 9 symbols of empty space.

**filled** checks the number of empty spaces and winners.

**winner** checks if symbol **p** is the winner.

**on\_move** updates the buffer: writes the symbol of the move to the move position.

**moves** returns indices of empty spaces.

Function **enumerate** returns an enumerated list, for example **enumerate([“a”,“b”])** returns **[(0,“a”),(1,“b”)].**

And last, function returns the text representation of a board.

Listing 5.1: Tic-Tac-Toe: Board

|  |
| --- |
| **class** board:  **def** \_\_init\_\_(self):  self.empty = " "  self.buffer = [self.empty]\*9  **def** filled(self):  **return** len([x **for** x **in** self.buffer **if** x is self.empty])==0 \  **or** self.winner("X") **or** self.winner("O")  **def** winner(self,p):  winning\_lines = [[0,1,2],[3,4,5],[6,7,8],  [0,3,6],[1,4,7],[2,5,8],  [0,4,8],[2,4,6]]  **for** line **in** winning\_lines:  **if** self.buffer[line[0]] == p **and** \  self.buffer[line[1]] == p **and** \  self.buffer[line[2]] == p:  **return** True  **return** False  **def** on\_move(self,move,symbol):  self.buffer[move] = symbol  **def** moves(self):  **return** [c[0] **for** c **in** enumerate(self.buffer) \  **if** c[1] **is** self.empty ]  **def** repr(self):  **return** ”{}|{}|{}\n−−−\n{}|{}|{}\n−−−\n{}|{}|{}”.\  format(\*self.buffer) |

**Assignment:** Change the function **winner**: Do not use hard-coded winning lines.

### Players

The next elements of the game are two players: the computer player-and the human player. The former must make moves based on some algorithm, while the latter accepts moves from a human player. Let’s assume the computer always makes the first move.

The base class:

Listing 5.2: Tic-Tac-Toe: Player, base class

|  |
| --- |
| **class** player:  **def** \_\_init\_\_(self,symbol):  self.symbol = symbol  **def** move(self,board):  **pass** |

We need only two functions: initialization (assigning a symbol) and making a move (doesn’t do anything in the base class).

Two derived classes:

Listing 5.3: Tic-Tac-Toe: Player-computer

|  |
| --- |
| **class** computer(player):  **def** \_\_init\_\_(self):  player.\_\_init\_\_(self,”X”)  **def** move(self,board):  options = board.moves()  **return** options[randint(0,len(options)−1)] |

Listing 5.4: Tic-Tac-Toe: Player accepting moves from a human

|  |
| --- |
| **class** human(player):  **def** \_\_init\_\_(self):  player.\_\_init\_\_(self,”O”)  **def** move(self,board):  **print** board  i = **int**(**input**(“>“))  **if** i **not in** board.moves():  **return** self.board.move(board)  **return** i |

Initialization functions specify symbols.

The computer player makes moves randomly. First it takes all available moves for the board and then returns the value at a random index.

The **move** function for a human player first prints the board and then waits for the human to type in a move. If the input is not in the list of available moves, it calls itself to again request the input from the player.

### The game

Finally, let’s consider the implementation of the game. We’ll need several functions:

* + - 1. Initialization: creating an empty board and initializing players.
      2. After each move we need to switch players.
      3. Functions that control the game: accepting a move from a player and updating the state of the game until the board is filled.
      4. At the end of the game we want to show the final state.

Here is the code:

Listing 5.5: Tic-Tac-Toe: The game

|  |
| --- |
| **class** TTTRandom:  **def** \_\_init\_\_(self, player1, player2):  self.player1 = player1  self.player2 = player2  self.board = board()  **def** switch(self):  self.player1,self.player2 = self.player2,self.player1  **def** play(self):  **while not** self.board.filled():  move = self.player1.move(self.board)  self.on\_move(move)  **def** on\_move(self,move):  self.board.on move(move,self.player1.symbol)  **if** self.board.filled(): self.done()  **else**: self.switch()  **def** done(self):  **print** ”Done:”  **print** self.board |

Review the code and confirm that it implements all the elements listed above. And this is the code to play the game:

Listing 5.7: Tic-Tac-Toe: Playing the game

|  |
| --- |
| p1 = computer ( )  p2 = human ( )  game = TTTRandom( p1 , p2 )  game . p l a y ( ) |

**Assignments:**

1. Combine the code for the board, players, and game. Play the game.
2. Change the **move** function to support “win and don’t let win” strategies.

## Tic-Tac-Toe: Minimax approach

As we can see, implementing a rule-based strategy can be a problem even for simple games like Tic-Tac-Toe. The board’s small size simplifies development, but any changes in rules or in size would require starting from scratch. At the same time, the number of possible moves is finite, and we can try to review all possibilities and to choose winning moves. The number of possibilities can be huge for games like chess, but for Tic-Tac-Toe we can consider all possibilities and build an optimal player.

The problem with the development is that we don’t know how an opponent will move. The opponent may follow a strategy we don’t know in advance. To address

the issue, we must assume that both players move optimally, trying to maximize the reward.

Consider the following example:



This is the tree of some game. The first player makes the first move (from **Start** to **2** or **4**), then the second player makes a move. The numbers associated with the graph nodes are the rewards.

Two players play against each other. The first player is trying to maximize the final reward: the reward at the end of the game after the second move. The first player is a **maximizer**. The second player is a minimizer and is trying to minimize the final reward.

Let’s see what a sequence of moves can look like:

* + 1. The first player moves to **4**. In this case the second player will choose **1**,as it’s the minimum.
    2. The first player moves to **2**. The second player will choose **5**.

Assuming that the second player chooses optimally, the first player should move to **2**. The final reward in this case is **5**,compared to **1** in the other case. The assumption of the optimal player is crucial here: For example, the second player choosing randomly may choose **10**.

The idea of the approach can be expressed in the following way: **A player must choose a move that minimizes the opponent’s maximum reward**. Remember that we have players of two types: minimizers and maximizers. A minimizer reads the rule as **a move that maximizes the opponent’s minimum reward**.

For games like Tic-Tac-Toe, the reward is known only at the end of the game: A player either wins or loses, or the game ends with a draw. The first player (X) is a maximizer, and in the case where it wins, the reward is +1. The second player is a minimizer, and the reward is -1. The reward at a draw is 0.

To apply this approach to Tic-Tac-Toe, we need to construct a computer player that analyzes the game tree (all possible moves) and each time chooses the best **minimax** move. Each move creates a new position on the board until the game is over. We can use recursion to build the game tree and analyze the results of a move.

The Python code is similar to the code we developed in the previous section, but we need to make several changes:

1. The class **board** needs a function that returns a reward depending on the state of the board.
2. The class **computer** needs an implementation of the **minimax** function and changes in the **move** function.

Here is the code:

Listing 5.8: Tic-Tac-Toe: The board, reward function

|  |
| --- |
| **class** board :  ...  **def** reward(self):  **if** self.winner("X"): **return** 1.0  **if** self.winner("O"): **return** -1.0  **return** 0.0  ... |

The function returns a non-zero reward depending on the winner (X or O) and zero in all other cases.

The **minimax** function (Listing 5.9) takes three parameters: **self** as any member function; **board**, the same as in the previous section; and **maximizing**, a boolean flag indicates a maximizer or minimizer (**True** for maximizer, **False** for minimizer).

Listing 5.9: Tic-Tac-Toe:Minimax computer player

|  |
| --- |
| **class** computer(player):  ...  **def** minimax(self,board,maximizing):  score = board.reward()  **if** score != 0: **return** score  **if** board.filled(): **return** 0  **if** maximizing:  best = -10000.0  **for** m **in** board.moves():  m\_board = deepcopy(board)  m\_board.on\_move(m,"X")  best=max(best,self.minimax(m\_board,not maximizing))  **else**:  best = 10000.0  **for** m **in** board.moves():  m\_board = deepcopy(board)  m\_board.on move(m,"O")  best=min(best,self.minimax(m\_board,not maximizing))  **return** best |

First, we get the current reward from the board. If the reward (**score**) is +1 or -1 (non-zero) the function returns the value. If the score is 0 and the board is filled (draw), the function returns 0. In all other cases we have moves to make. Each move returns a new score, and we must choose the best score. The notion of the best depends on the **maximizing** flag: It is **max** for a maximizer and **min** for a minimizer. Note that we are flipping the **maximizing** flag: If the current player is a maximizer, the next move makes the player a minimizer, and vice versa. When calling **self.minimax** recursively, we use the expression “not maximizing” to change the type of player.

And the **move** function (Listing 5.10). The list **mm** consists of minimax values of each possible move. Note that the call of **self.minimax** is for the minimizer: The flag provided to the function is **False**. This is because the computer player is a maximizer and the next move is for a minimizer.

The function creates a list of all minimax values and returns the move that corresponds to the maximum of minimax. If the same minimax value appears more than once, the function returns the first move associated with the value.

Listing 5.10: Tic-Tac-Toe: Computer player, move function

|  |
| --- |
| **class** computer(player):  ...  **def** move(self,board):  moves = board.moves()  mm = []  **for** m **in** moves :  m\_board = deepcopy(board)  m\_board.on\_move(m, self.symbol)  mm.append(self.minimax(m\_board,False))  mm\_index = mm.index(**max**(mm))  **return** moves[mm\_index]  ... |

**Assignments:** Adjust the code from the previous section to incorporate changes for the board and computer player. Play the game. You may find that the first move takes time; that is expected, and we’ll address this problem in the next section.

The approach doesn’t distinguish between cases when the player can win in one or five moves. You may find that sometimes the computer will not choose the quickest possible strategy. The reward is always the same (+1 or -1) and doesn’t depend on the number of moves. We can address this issue by penalizing moves.

The recursive call of the **minimax** function can be changed by adding the depth of recursion as a parameter. Consider as an example the function that returns the factorial of a number:

Listing 5.11: Factorial calculations

|  |
| --- |
| **def** factorial(n):  **if** n <= 1: **return** 1  **return** n ∗ factorial(n−1) |

We can add depth in the following way:

Listing 5.12: Factorial calculations with depth printing

|  |
| --- |
| **def** factorial(n, depth):  **print** depth  **if** n <= 1: **return** 1  **return** n ∗ factorial(n−1,depth+1) |

Try to call this function as **factorial(5,0)**: it will return 5! (120) and will print the depth of each function call starting from 0.

The depth is the number of recursive function calls. In the case of the **minimax** function, the depth is the number of moves: Each call corresponds to a move. One way to take into account the number of steps is to reduce the reward by the number of steps. Now the reward of the maximizer is +1. You can change the reward to 10 and subtract the depth:

Listing 5.13: Tic-Tac-Toe: Code correction to take into account the number of steps

|  |
| --- |
| **def** minimax(self,board,maximizing,depth):  ...  best = **max**(best,self.minimax(m\_board,not maximizing,\  depth +1))  ...  **return** best − depth |

This code penalizes solutions with bigger number of steps. In this case the winning strategy that requires the least number of moves will be prioritized.

**Assignment:** Change the code of the **minimax** function to take into account the number of moves as discussed above.

## Tic-Tac-Toe: Q-learning

This section introduces the advanced topic of Reinforcement Learning: The technique in one form or another is used in a wide range of applications, from developing an invincible chess program to teaching a robot to perform a backflip.

Tic-Tac-Toe is a game with a small number of possible moves and a short horizon: The game ends in a maximum of 9 moves. This simplifies the development of the learning algorithm. The method we’ll use is called Q-learning, and it’s rooted in Bellman’s Principle of Optimality. If we have an optimal sequence of decisions—for example, winning moves in Tic-Tac-Toe—then at each moment the remaining part of the sequence is also optimal. If the sequence of moves  is an optimal one, then the sequence *mi*, *mi+1*, . . . *mN* is also optimal starting from any moment *i*. In other words, the optimal sequence doesn’t depend on previous steps.

Let’s consider the sequence of actions (moves) *ai* and the associated sequence of states *si*. States are results of actions: For Tic-Tac-Toe, states are boards after a sequence of moves, and each new move changes the board:

Each state *si* is associated with a reward *ri*. For Tic-Tac-Toe, the reward is zero for all states except winning positions.

We want to find an approach (often called a policy *π*) that maps all possible states *S* to all actions *A*:

*π* : *S → A*

This policy gives us an action for a state. Our goal is to create a policy that maximizes future rewards: that is, it maximizes our chances to win in a game.

Let’s introduce future rewards *V* as a sum of all rewards associated with future states:

*V* (*si*) = *ri* + *ri*+1 + *ri*+1 + *. . .*

It is more common to discount future rewards:

*V* (*si*) = *ri* + *γ·ri*+1 + *γ*2*·ri*+2 + *...*

where *γ* is a number from 0 to 1, normally closer to 1. *V* depends on the policy *π*. We denote *V ∗* as the maximum possible future reward.

For the state *s* action *a* creates new state *sI*:

Let’s introduce function *Q*(*s, a*):

The function *Q* is the sum of the current reward plus the discounted maximum future reward. Taking into account that

we can write

This gives us a recursive definition of *Q*.

We can use this definition to find an approximation for *Q*. We’ll start with some approximation *Q*1: For example *Q* = 1 for all states and actions. If we are lucky, and *Q*1 is the exact solution, we’ll have

But most probably the difference between the right-hand and left-hand sides of the equation is not zero:

Let’s update our approximation of *Q* in the following way:

Parameter *α* is a small number that controls the rate of updates. This gives us the next approximation, and we’ll find ∆*Q*2 and the next approximation of Q:

Each step updates our approximation of *Q*. At the step *n* we have the next approximation, *Qn*+1:

If we run this procedure long enough, we may notice that values of ∆*Q* become smaller and smaller: The process converges to some *Q[[4]](#footnote-4)*.

Let’s use this approach to create (to learn) a *Q*-function for a Tic-Tac-Toe player. The player then will choose the move that maximizes *Q* for any given state (board).

The question is how to create an iterative procedure as described above. We can, for example, play against the computer, and it will eventually learn the results of moves at various positions of the board. This is possible but may take a lot of time and human effort: The number of possible moves is large, even for a simple game like Tic-Tac-Toe.

The alternative approach is to let the computer play against itself to learn the *Q*- function and use the resulting function to play against a human. We’ll need to implement two modes of operation in the game: learning and playing. During the learning mode, two computer players share the same *Q*-function, choose the best moves, and update the *Q*-function according to the resulting rewards. We can let them play hundreds of thousands of times to learn the *Q*-function. To let the player explore new positions, we’ll make random moves as well. When playing against a human, the computer player will use the learned *Q*-function to choose the best moves.

The scheme described above requires several changes in the code of the Tic-Tac-Toe game.

The changes will affect all elements. An important addition is the variable **Q**: the mapping from the state to all possible moves and rewards associated with those moves. **Q** is introduced as a dictionary where keys are states and values are maps (dictionaries) from moves to rewards.

The board needs a function that returns the state of the board: the buffer and the symbol of the active player (Listing 5.14).

Listing 5.14: Tic-Tac-Toe: Q-learning, board

|  |
| --- |
| **class** board:  ...  **def** state(self,Q,symbol):  state = "".join(self.buffer) + symbol  **if** state **not in** Q:  Q[state] = {m:1.0 for m in self.moves()}  **return** state  ... |

The variable **state** is constructed as a string representing the buffer with a symbol of an active player added to the end. Note that adding the symbol is redundant, as the player can be identified from the board, assuming that “X” makes the first move. Each time we find a new state, we need to update **Q**: We are creating a new key and assigning default (1*.*0) rewards to each available move.

Listing 5.15: Tic-Tac-Toe: Q-learning, computer player

|  |
| --- |
| **class** computer:  **def** \_\_init\_\_(self, symbol,Q={}, eps =0.2) :  self.symbol = symbol  self.Q = Q  self.eps = eps  **def** move(self,board,learning):  moves = board.moves()  **if** random() < self.eps **and** learning:  **return** moves[randint(0,len(moves)-1)]  state = board.state(self.Q,self.symbol)  qs = self.Q[state]  minmax = max  **if** self.symbol == "O" : minmax = min  mm = minmax(qs.values())  options = [**int**(m) **for** m **in** qs **if** qs[m]==mm]  **return** options[randint(0,len(options)-1)] |

The next change is in the implementation of the computer player. The changes start from the initialization. Here is the new implementation:

Initialization now includes assigning two new variables: **Q** and **eps**. **Q** is the function we want to learn. During the learning, the function is shared between two computer players so that each of them can make adjustments based on results. Variable **eps** controls the rate at which the player can choose a random move in learning. If **eps=0.2** (default value), then the player makes on average one-fifth random moves.

A function **move** also takes a new parameter, **learning**: a boolean flag specifying that the player is not playing against a human, but learning the function **Q**.

The function **move** first takes all possible moves, and if conditions are met (learning mode and random number are in the expected range[[5]](#footnote-5)), it returns a random move from the list of all possible moves. In the other case, the function takes the state and extracts the map from moves to expected rewards (variable **qs**).

As discussed, there are two types of players: maximizers (“X”) and minimizers (“O”). Depending on the type, we need to choose the move that corresponds either to the maximum reward in **qs** or to the minimum reward. To make the code a little more compact, we are using the fact that functions in Python can be treated as variables: We initialize the **minmax** variable by assigning it (max) (and this is the internal Python function!) or (min) if the symbol of the player is “O” (minimizer). The last step is to identify a move that corresponds to the max (or min for minimizer) reward value. Note that if we have more than one move with the same reward, we’ll take the first in the list.

Now let’s review changes in the implementation of the game.

**TicTacToeGame** class intialization:

The function **init** takes two players as parameters and the new variable **Q**: the mapping that represents the *Q*-function. By default, the game starts with an empty mapping: This is the case when we start the learning process and don’t have the *Q*-function. We can provide a pretrained **Q** to play the game against a human player.

Over the course of the game, we’ll switch players. **self.player1** is always the player that makes the next step. When learning, we’ll be restarting the game, and we need to start with the same player. The variable **self.starting symbol** is needed to distinguish two players after a sequence of moves.

The variable **self.learning** is the flag indicating that we are learning the *Q*- function: the case when the computer player competes with the other computer player.

As before, we are initializing an empty board.

Two new variables, **self.gamma** and **self.alpha**,control the iterative process and correspond to *γ, α* as discussed above.

The last part of initialization is assigning **Q** to computer players: When learning, they share the same function.

Listing 5.16: Tic-Tac-Toe: Q-learning, game initialization

|  |
| --- |
| **class** TicTacToeGame :  **def** \_\_init\_\_(self, player1, player2, Q={}):  self.starting\_symbol = player1.symbol  self.player1 = player1  self.player2 = player2  self.learning = isinstance(player1,computer) **and** \  isinstance(player2,computer)  self.Q = Q  self.board = board()  self.gamma = 0.9  self.alpha = 0.3  **if** isinstance(self.player1, computer): self.player1.Q = Q  **if** isinstance(self.player2, computer): self.player2.Q = Q |

The function **restart** first ensures that the symbol of the first player corresponds to the starting symbol and switches players if needed. Then it initializes the new empty board.

Listing 5.17: Tic-Tac-Toe: Q-learning, the game restart

|  |
| --- |
| **class** TicTacToeGame:  ...  **def** restart(self):  **if** self.starting\_symbol != self.player1.symbol:  self.switch()  self.board = board()  ... |

The only addition to the **on move** function is learning the move: the first two lines.

Listing 5.18: Tic-Tac-Toe: Q-learning, on move

|  |
| --- |
| **class** TicTacToeGame:  ...  **def** on\_move(self,move):  **if** self.learning:  self.learn(move)  self.board.on\_move(move,self.player1)  **if** self.board.filled():  self.done()  **else:**  self.switch()  ... |

Let’s look at this core function in detail.

Listing 5.19: Tic-Tac-Toe: Q-learning, learning a move

|  |
| --- |
| **class** TicTacToeGame:  ...  **def** learn(self,move):  state = self.board.state(self.Q,self.player1.symbol)  new\_board = deepcopy(self.board)  new\_board.buffer[move] = self.player1.symbol  reward = new\_board.reward()  **if not** new\_board.filled():  new\_state = new\_board.state(self.Q,self.player2.symbol)  qs = self.Q[new\_state]  **if** self.player1.symbol == "X":  reward = self.gamma\*min(qs.values())  **else**:  reward = self.gamma\*max(qs.values())  self.Q[state][ move ] += \  self.alpha\*(reward-self.Q[state][move])  ... |

First, we construct the current **state**: The board returns the string representation. To evaluate the move, we first make a copy of the board and assign the symbol to the specified move (the third line of the function can be substituted with **new\_board.on move(move,self.player1)**). After the move we get the **reward**.

We’ll accept this reward only if the move was the last move (and we know the result of the game) for the board. Before we review what we do otherwise, let’s note that if **player1** is a maximizer (X), then after the move the state corresponds to the player-minimizer, and vice versa.

If the board is not filled, we’ll need to use as a reward the discounted value of the minimum (for the player-maximizer; see the note above) or maximum (for the minimizer) **Q.** We do this by first extracting all possible moves with their associated rewards—variable **qs**—for the state of **new board**. Then we choose the max/min reward and discount it by multiplying by **self.gamma**.

The last step is to update **Q** according to the formula discussed in the beginning of the section.

**Python:** Learn how to save Python structures in JSON files.

Now we need to train the model:

Listing 5.20: Tic-Tac-Toe: Q-learning, training, computer against computer

|  |
| --- |
| player1 = computer("X")  player2 = computer("O")  game = TicTacToeGame(player1,player2)  **for** \_ **in** range(100000):  game.play()  game.restart()  **with** open("ttt.json", "w") **as** f:  json.dump(game.Q,f) |

Two computer players play 100,000 games updating the *Q*-function. When done, the trained variable is saved for future use in a JSON file.

How to play: We first load **Q** from the file and then provide it as a parameter when initializing the game: The computer player will be using this pre-trained function.

Listing 5.21: Tic-Tac-Toe: Q-learning, human against trained computer

|  |
| --- |
| **with** open("ttt.json", "r") **as** f:  Q = json.load(f)  player1 = computer("X")  player2 = human("O")  game = TicTacToeGame(player1, player2 ,Q)  game.play() |

**Assignment:** Combine the pieces of code discussed above with previous implementations of Tic-Tac-Toe. Train the model. Load the model and play against the computer. Note that multi-step training may take time.

## Challenge

In this chapter we’ve learned how to develop three types of Tic-Tac-Toe players using a rule-based approach, minimax, and Q-learning. The challenge now is to compare them among each other and their implementations by groups of students.

* + 1. Implement rule-based Tic-Tac-Toe player. Start with “win, don’t let win” strategy. Expand it by adding new rules: Use your experience to come up with a winning strategy. Make the code reusable, so that computer players developed by groups of students can compete.
    2. Implement the game where a rule-based player plays against a minimax player.
    3. Implement the game where a minimax player plays against a Q-function player.
    4. Experiment with parameters used to create a Q-function (number of games, *γ, α, c*). Implement the game where two players with different *Q*-functions play against each other, and run a competition of models generated with different parameters.

# Chapter 6

**Artificial Neural Networks**

## Background and basic ideas

This section will provide a very basic introduction to the field of Artificial Neural Networks (ANN or NN): a technology with many decades of history and which has been growing actively in the last 10 to 15 years. Advances in high-performance computing and easy access to a variety of datasets have made NN one of the central technologies in machine learning and artificial intelligence.

As the name suggests, NNs are in some way related to the structure of a brain. The history and relationships between developments in AI and in neuroscience are not subjects of this course, but we’ll see that NN structure and operations resemble two elements of biological neural systems:

* + 1. Threshold-like activation potential: For a neuron to “fire,” the electrical potential must exceed some level.
    2. Neurons are interconnected: They form a complex network.

The ideas on NNs in artificial intelligence came only partially from biology.[[6]](#footnote-6) The main area where NN principles are rooted is the question of computability, the fundamental problem in computer science. Turing, von Neumann, and Kolmogorov are among many others who structured the field.

Let’s now formulate a basic problem. We have an input—a numerical value, a list of numerical values, an image (can be presented as a list of numbers), etc.—and an associated output. For example, the input can be an image and the output a Boolean value indicating that the image represents a cat. We want to build a model *F* that can associate an output to a given input. For the example above: to find all images with cats.

In other words, for a set of inputs *X* we want to find a function *F* that maps *X* into a set of outputs *Y*:

*F* : *X → Y*

Or for multi-dimensional *X* (*xi, i* = 1*..N* ) and *Y* (*yi, i* = 1*..M* ):



In this section we’ll consider only one type of NN: multi-layer perceptron, the generalization of single-layer perceptron. Let’s start with the latter and first assume that there is only one output value (this doesn’t affect the discussion, as we can assume that we have an individual function *F* for each output component):



### Single-layer perceptron

The single-layer perceptron scheme is:



We first calculate a weighted sum of input components and then calculate function *f* (*·*) of the sum. The choice of *f* is crucial. It’s easy to see that the linear function *f* (*x*) = *a · x* + *b* is the equivalent of linear dependence between *X* and *Y*:



where and .

The common choice is a smoothed version of a step function (sigmoid). The step

function:



Sigmoid:



The sigmoid is close to the step function for big positive and negative values of *x*, and smooth behavior at *x* = 0 makes it convenient for a number of applications.

One of the other options is a RELU (REctified Linear Unit):



This is not the complete list of functions used in NN, as there exist functions with more complex and not always deterministic behavior, but the idea remains the same: some nonlinear and often threshold-like function.

Let’s see how this works for simple dependencies between *X* and *Y*. For the exercise we’ll choose a step function and basic logical operations (NOT, AND, OR, XOR), as they are simple enough to implement without coding to get an idea of the approach.

**Logical NOT**. Here is the table of possible cases[[7]](#footnote-7):

X NOT X

1 0

0 1

Consider the following weights:

*w*0 = 0*.*5*, w*1 = *−*1

*Y* = *f* (*−X* + 0*.*5)



As we can see, *Y* corresponds to *NOT X*.

**Logical AND**. The same steps:



**Logical OR**. The same steps:



**Assignment:** This case is similar to AND; find *W*0*, W*1*,* and *W*2.

#### Logical XOR (exclusive OR)



Here we have a problem: A single-layer perceptron cannot compute XOR for all four combinations of arguments. Let’s see what’s different between XOR and the other two variable functions (AND, OR). To do this we’ll represent functions in the following way:



We can find that zeros and ones in AND and OR form “islands” that can be separated by a single line: Just apply a ruler, and you will find that you can have all zeros in AND above the ruler and one below the ruler. The same can be done for OR:



XOR is different: You need at least two lines to separates zeros and ones:



The linear separability is an important feature. It can be introduced in a formal way, and it’s possible to prove that a single-layer perceptron cannot implement a function that is not linear-separable.

### Multi-layer perceptron

Perceptron structure can include a hidden layer or layers.



Calculations of output values include two steps: First we calculate outputs of hidden layer nodes and then use them to calculate output. The output *zk* of the *k*-th node of the hidden layer (*k* = 1*..Nh*):

where is the weight of connection between *i*-th input node and the *k*-th hidden layer node. Output values *yj* are:

Note that the upper index 2 in doesn't indicate the square of the number. Rather, it refers to the second set of weights—weights between hidden layer nodes and output nodes.

We can create NN with more than one hidden layer, but the idea remains the same: we want to build a layered structure where the output of a node is a function of the previous layer nodes’ weighted sum of outputs. Those weights have to be calculated in the process known as “learning.” Given the set of input/output values, we want to choose weights to minimize the total error of the outputs calculated. For example, if we have 1,000 images and 500 depict cats, we want to train an NN that can correctly identify most of the pictures with cats[[8]](#footnote-8). To do this, we can introduce a formal measure of the error that depends on the set of inputs () and outputs () and all weights (*W*) ():



Learning is the procedure of finding weights *W* that minimize the error:

The common approach to NN training is called back-propagation. The concept and implementation of this approach are out of the scope of this course as they require understanding of basic optimization techniques. Instead we’ll be using programming libraries where all required functionality is available without going deep into the theory.

We’ll consider only the multi-layer perceptron – the course doesn’t cover such important and interesting cases as Convolutional NN (CNN), Recurrent NN (RNN), Long Short-Term Memory (LSTM), and Hopfield networks (RNN with binary node states). Model-specific modifications do not change the idea of the learning: We want to choose weights and sometimes other parameters to minimize the output approximation error.

## Neural networks and rules of “Life”

In the previous sections we were analyzing systems with known behavior: mazes, 8-puzzles, and Tic-Tac-Toe games. Now let’s try to identify rules from observations. We’ll be looking at some system and trying to replicate its behavior without knowing the underlying rules.

“Life,” the game introduced by Conway, has simple rules resulting in interesting behavior. “Life” is an example of cellular automata: systems that consist of cells on a rectangular grid with the behavior of a cell defined by its neighbors:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  | *·* | *·* | *·* |  |
|  | *·* |  | *·* |  |
|  | *·* | *·* | *·* |  |
|  |  |  |  |  |

Each cell can be alive or dead. A cell changes state depending on its own state and the state of 8 neighbors: The rules of how cells change state are:

* + 1. A live cell remains a live cell if it has two or three live neighbors; otherwise it dies.
    2. A dead cell becomes a live cell if it has three live neighbors: otherwise it remains dead.

Here are several examples of the static structures, which do not change:

**Block**: Each live cell has three live neighbors, while each dead cell has no more than two live cells:



**Boat** (confirm that this configuration is a static one):



Configurations with periodic behavior:

#### Blinker

#### 

#### Toad

#### 

**Assignment:** Write a function returning new state for the central cell on 3x3 grid

A cell on a boundary has only 5 neighbors, which may make it die faster than cells inside:



To avoid this problem, we can introduce periodic conditions: We’ll assume that neighbors of a cell on a border also include three cells on the opposite border:



In this case, the following starting configuration doesn’t disappear (live cells have only one live neighbor without taking into account cells on the opposite border), but becomes an equivalent to the “Block” (see above).



The same conditions can be applied to left and right borders.

**Assignments:**

1. Write a Python function that returns the new state for a cell given by coordinates (i,j) on a board with periodic conditions.
2. Write a Python function that updates a board of an arbitrary size. Assume periodic conditions and add a function that prints a board (use the space character for a dead cell and an asterisk for a live cell).

## Learning the rules of “Life”

The goal of this section is to train a NN that can derive rules of Conway’s “Life” from observations. To begin with, we’ll assume that we know something about the system: Namely, we are assuming that the change of a cell’s state depends on its closest neighbors, but we are not making any assumption about the dependence. Later we’ll consider the case where we are observing “Life” on a big board without making any assumption about dependencies between cell states.

To solve the first problem (reproducing the state change of an individual cell), we’ll do the following:

* + 1. Given a 3x3 board, we’ll generate all possible configurations of dead/live cells. The set of all configurations is our input.
    2. The output is the new state of the central cell for any input. We do not consider any other cell except the central one: We’ve made the assumption about the dependence on neighbors that makes this possible, and it’s enough to find rules for the central cell only.
    3. We have a 9-component input and a single component output. We’ll construct an NN, train it, and test the accuracy.

As we’ve seen in the past, we can use a list of 9 numbers (zeros and ones) to represent the board 3x3. For example, the list [1,0,1,0,0,1,0,1,0] corresponds to:

101

001

010

Ones correspond to live cells and zeros to dead cells.

Here is how we can create all possible combinations of 0s and 1s for a list of 9 elements. This code uses function **permutations** from the Python module **itertools[[9]](#footnote-9)**. In the loop we are creating a list of 9 elements with *n* zeros and 9*n* ones. For *n* from 0 to 9, calculate all possible permutations, and add the new combinations of zeros and ones to the list **all boards**. Think how else you can do this and how many configurations are possible.

*−*

Listing 6.1: All 0-1 combinations on a board 3x3

|  |
| --- |
| **from** itertools **import** permutations  **from** random **import** shuffle  all\_boards = [ ]  **for** i **in** range(10):  x = [0]\*i + [1]\*(9-i)  all\_boards += list(set(permutations(x,9)))  shuffle(all\_boards) |

The last step shuffles the list of boards to randomize the order of elements. Later we’ll be splitting this list, and we don’t want to have any special order. Before we call the **shuffle** function, the list **all boards** is ordered by the number of zeros; the first element doesn’t have zeros, next go elements with 1 zero and 8 ones, then with 2 zeros and 7 ones, etc.

**Assignment:** The code above is calling **set()** to extract all unique elements from the output of **permutations** function. Experiment with **permutations** functions to learn how it works for lists and explain why the output of **permutations** contains non-unique elements.

Now we need to generate an output for each input:

Listing 6.2: New state for the central element of a board 3x3

|  |
| --- |
| **def** new\_state(L):  # number of live neighbors (all elements except the central one)  sum\_nbrs = sum(L[:4]) + sum(L[5:])  **if** (L[4] == 1 and sum\_nbrs in [2,3]) **or** \  (L[4] == 0 and sum\_nbrs == 3 ): **return** 1  **return** 0 |

Applying this function to all input boards:

Listing 6.3: New states for all boards

|  |
| --- |
| new\_centeres = [new\_state(x) for x in all\_boards] |

**Python:** Learn about the **scikit-learn** library and how to use it to train a multi-layer perceptron.

Now we have all inputs and all outputs and can train the NN (Listing 6.4). We are creating **numpy** arrays **X** and **Y** from inputs and outputs, initializing NN (variable **nn**), training NN, and getting predicted outputs (**Y predicted**). Outputs are not zeros and ones, but we can create this representation using 0.5 as a threshold: Output below 0.5 is considered to be 0 and above 0.5 corresponds to 1.

Listing 6.4: Training the NN

|  |
| --- |
| X = np.array(all\_boards)  Y = np.array(new\_centers)  nn = MLPRegressor( hidden\_layer\_sizes =(9),  activation='logistic',  solver='lbfgs' )  n = nn.fit(X,Y)  Y\_predicted = nn.predict(X).to\_list()  predicted\_centers = [(1 **if** c>0.5 else 0) **for** c **in** Y\_predicted] |

**Assignments:**

1. Combine all code elements discussed above: generation of all possible boards 3x3, calculation of new states of the central element, NN training, and calculation of predicted centers. Run the program to confirm it works.
2. Add calculation of the total error: the number of cases when centers used to train the NN (**Y**) do not match predicted centers.
3. Experiment with the NN structure: Change initialization parameter **hidden layer sizes** and analyze its influence on the code’s accuracy.

We have used all possible inputs and all possible outputs to train NN. This is a valid approach, as we know that no other input is possible. For many other applications, we have observations that do not cover all possible inputs/outputs. In this case, the model trained on known data will be applied to unknown inputs. We can emulate this condition by splitting our observations (all inputs and outputs) into two parts: for training and for testing.

Listing 6.5: Splitting data into training and testing datasets

|  |
| --- |
| X\_training = np.array(all\_boards[:400])  X\_testing = np.array(all\_boards[400:])  Y\_training = np.array(new\_centers[:400])  Y\_testing = np.array(new\_centers[400:])  nn = MLPRegressor( hidden\_layer\_sizes =(9),  activation='logistic',  solver='lbfgs')  n = nn.fit(X\_training, Y\_training )  Y\_training\_predicted = nn.predict(X\_training).to\_list()  Y\_testing\_predicted = nn.predict(X\_testing).to\_list()  predicted\_centers\_training = \  [(1 if c>0.5 else 0) for c in Y\_training\_predicted]  predicted\_centers\_testing = \  [(1 if c>0.5 else 0) for c in Y\_testing\_predicted] |

We are taking first 400 inputs/outputs to train the NN and the remaining inputs/outputs to test the model.

**Assignment:**

1. Change the program from the previous assignment to support splitting data into training and testing datasets.
2. Add calculation of accuracy separately for training data (so called in-sample accuracy) and for testing data (out-of-sample).
3. Experiment with relative sizes of training and testing datasets. Change “400” from the code above to other values and analyze how in-sample and out-of-sample accuracies change.

Splitting known data into training and testing datasets is a very important technique: It’s used to prevent overfitting. If you have a model with many parameters, you can have high in-sample accuracy and low out-of-sample accuracy.

Choosing the model structure (the number of elements in hidden layers in the case of NN) and using a training procedure that reduces chances of overfitting the analysis of relative accuracy in- and out-of-sample are widely used not only for NN, but for virtually all machine learning techniques.

## Challenges

In this chapter you’ve learned the basic ideas of using neural networks to reproduce the rules of the Game of Life from observations of the central cell state changes on the board 3x3. The challenge now is to apply the same approach on a bigger scale.

* + 1. Implement the Game of Life on a big board (10x10 or bigger) with periodic boundary conditions. Prepare two versions: one based on the original rules and the other on rules generated by trained NN. Confirm that for any initialization these two versions produce the same configurations.
    2. Implement the Game of Life on a big board (start with 10x10) with periodic boundary conditions. Train NN that takes as an input the configuration of the entire board and outputs the configuration of the entire board as the next step. For the 10x10 board both input and output are 100-dimensional vectors. Create a training dataset by randomly initializing the board and making one move. Create a testing dataset in the same way. You can compare the accuracy of your solutions by comparing results of applying trained models to a dataset specially generated for the competition.
    3. The same as above, but now the output is the board after **two steps.** To generate the dataset, you first need to initialize the board (this configuration is your input) and then make two steps (the resulting configuration is your output).

# Appendix A Python basics

This section covers elements of Python programming required to complete the course. Please refer to a systematic course on Python to learn more advanced elements of the language.

## Programming: Basic concepts

Let’s start with reviewing elements that don’t depend on the programming language and then see how they work in Python. Programs implement algorithms and as a rule take some data as an input and generate data as an output. Data types are domain-specific: Different applications require different data elements and data organization, but the most common types are:

* + 1. Logical constants: True and False.
    2. Single numbers: Last market price, number of cars in a parking lot, age, number of seconds between two events, universal constants, etc. 5, 3.1415926, 123.97.
    3. Text: Single characters, words, sentences, documents, text of news.
    4. Lists of elements: Historical prices, weather records, network traffic. Elements of lists can be accessed by their position in the list: first element, second element, etc. In Python, the first element is element 0. Lists are very common and fundamental in programming and in CS. The name of one of the oldest programming languages—LISP—stands for LISt Processor.

## A few words on programming environments

If Python is not installed, please go to python.org and follow instructions for your operating system. We assume a student can create a text file with the code of the program and execute the command to run it.

Traditionally, the first program prints “Hello world!”. Here is its listing in Python:

|  |
| --- |
| print "Hello world!" |

**Assignment:** Create file HelloWorld.py with this program and run in the command line:

Listing A.3: Python: run HelloWorld.py

python HelloWorld.py

## Basic operations

Start Python interactive shell and enter commands one by one. Lines started with # are comments, they are not executed. Create a file with commands and execute it.

|  |
| --- |
| # Boolean  b = True  # Integer number  x = 2  # Float number  y = 1.23  # String  s = "Abc"  # List of numbers:  L = [ 1 , 2 , 3 , 4 , 5 ]  # print everything:  print b , x , y , s , L  # Operations:  b2 = not b  print b, b2  x2 = x\*31  print x, x2  y2 = y\*y  print y, y2  s2 = s.lower()  print s, s2  L2 = L[::-1]  print L, L2  a = 2  b = "A"  print a,b  a,b = b,a  print a, b |

## Functions in Python

We use “def” to declare a new function, list parameters in parentheses, and column “:” at the end of a declaration. For example:

|  |
| --- |
| **def** f(x): **return** x\*x |

Here we’ve declared function **f**. The function takes one parameter **x**, and returns the square of the parameter. The function can fail if multiplication is not defined for the parameter **x**: if x is a string, for example.

The function can have more than one line of code. We can rewrite the function from our example in the following way

|  |
| --- |
| **def** f(x):  x\_squared = x\*x  **return** x\_squared |

Note indentation is usually four spaces. The indentation defines the body of the function and affects “visibility” of variables. Consider the following code:

|  |
| --- |
| x = 1  **def** f(x):  x2 = 2\*x  **return** x2  **def** g():  x2 = 2\*x  **return** x2  print f(2)  print g() |

First we’ve initialized variable **x** (line 1). Then we’ve defined two functions:

* + 1. **f** that takes a parameter with the same name **x** and returns **2\*x**. The variable **x** initialized on line 1 doesn’t affect this function, and on line 4 the variable **x** known to the function **f** is equal to 2.
    2. **g** doesn’t take a parameter and returns **2\*x** for any **x** defined before the function call. In this case **g** takes **x=1** defined on the line 1.

Functions in Python can be nested; one function can be defined in the body of another function:

|  |
| --- |
| x = 1  **def** f(x):  x2 = 2\*x  **def** g(x):  x = x+1  **return** x  **return** x + g(7)  print f(2) |

There are three independent variables **x** here. Note the double indentation—8 spaces—for the body of function **g**.

**Assignment:** What are values of **x** when you are on the lines 1, 4, or 6? Add print statements to show the value of **x** in different parts of the code to check your answer.

## Conditions

If your program needs to change behavior depending on some condition, you can use an **if** statement. Let’s say you want to print numbers spelled out as words. One simple way of doing this is to check the number and print accordingly.

|  |
| --- |
| n = 1  **if** n == 1:  print "one"  **elif** n == 2:  print "two"  **else**:  print "not one or two" |

Here we are using a double equals sign to check the value.

* + First we compare **n** with 1. If n is equal to 1, we print the word “one.”
  + If it’s not, we compare **n** with 2 (note the use of **elif** statement) and print accordingly.
  + For all other cases (**n** is not 1 or 2) we use the keyword **else**.

## Loops

There exist cases when your program has to iterate over elements of a list or a dictionary, or execute some part of a program several times, probably with different parameters. In such a case we can use the **for** statement:

|  |
| --- |
| L = [ "a", "b", "c", "def" ]  **for** e **in** L:  print e |

It is very common for your program to iterate over the list elements using their indices. The index is the position of an element in the list:

|  |
| --- |
| L = [ "a", "b", "c", "def" ]  **for** i **in** range(4):  print L[i] |

**range(4)** is a function returning a list of integers from 0 to n-1. The code above is equivalent to

|  |
| --- |
| L = [ "a", "b", "c", "def" ]  **for** i **in** [0,1,2,3]:  print L[i] |

Note that indices start from 0: The index of the first element is 0, the index of the second element is 1, etc. For the list of N elements the index of the last element is N-1.

## Exercises for A.1-A.6

1. Write function that returns sum of all elements in a list. Assume that all elements of the list are numbers. Write four versions:
   1. Use loop and indices of elements
   2. Use “for each” loop
   3. Use Python function
   4. Use recursion
2. Write function that returns first N Fibonacci numbers (N is a parameter). Write two versions:
   1. Use function from the course returning n-th Fibonacci number and populate a list.
   2. Use a single loop over pre-allocated list (create first a list of N ones).
3. Write function that returns roots of a quadratic equation . Coefficients a, b, c are parameters. Notes:
   1. You’ll need function that returns square root of a number: import it from standard Python module:

|  |
| --- |
| **from** math **import** sqrt  print sqrt(2.0) |

* 1. Suggest your own way to treat complex numbers. Do not use standard support of complex numbers in Python.
  2. Suggest your way to treat a case when .

1. Write function that reverses a list: for example, returns [3,2,1] for [1,2,3]. Write three versions:
   1. Create a new list and use a loop with indices
   2. Use “for each” loop in a reverse order
   3. Reverse “in place” – do not create a new list, but swap elements of the existing one.
   4. Use standard Python list operations.
2. Find the index of a max element in a list. Write two versions:
   1. Use a loop
   2. Use standard Python functions (tip: you’ll need functions **max** and **index**)
3. Write a function that checks if a string is a substring of another string. For example, it returns True for “ample” in “example”, and False for “cat” in “elephant”. Write three versions:
   1. Use a loop
   2. Use recursion
   3. Use standard Python functions
4. Write function that returns all elements of a list with odd indices (or even indices). Write three versions:
   1. Use a loop
   2. Use recursion
   3. Use standard Python list operations

## File input/output

### Text files: Reading/writing

Text files of different formats—unformatted plain text, comma-separated (CSV), JSON, XML—are widely used for a number of applications, and we’ll start with reading and writing text.

A file is referenced by its name. The file name may include a directory path. To start working with a file, you need to open it. Function **open** takes two parameters - file name and mode of operation. The mode **“r”** opens file for reading starting from the very beginning. The **“w”** mode means that the file is open for writing. If the file doesn’t exist, it will be created. If the file exists, it will be opened as an empty file, and the existing content will be lost. The **“a”** mode means that the file is open for appending. If the file exists, it will be open for writing to the end of the file.

The function **open** returns an object (variable **f** ) used to actually read/write. After reading/writing is done, the file must be closed with function **close**.

|  |
| --- |
| # create a file with one line of text inside:  f = open("filename.txt","w")  f.write("Hello world!")  f.close()  # read the file and print its content :  f = open("filename.txt","r")  content = f.read()  f.close()  print content  # append some text to the file  f = open("filename.txt","a")  f.write("Hello again")  f.close() |

**Assignments:** Experiment with opening files in different modes and reading/writing. Be careful with modes of operation.

### Command line arguments. Modules in Python.

To open a file, you need to provide a filename. This can be done directly in the code (hardcode the name). In this case you’ll need to change your program with every new file you use. As an alternative, you can specify the name as a parameter in the command line:

**python yourprogram.py filename.txt**

The code above can be rewritten in the following way:

|  |
| --- |
| **import** sys  f = open(sys.argv[1],"w")  f.write("Hello world!")  f.close()  f = open(sys.argv[1],"r")  content = f.read()  f.close()  print content  f = open(sys.argv[1],"a")  f.write("Hello again")  f.close() |

The first line contains a new element: **import sys**. Python has a notion of a module—its way of organizing code. To start using a module, it has to be imported.

Module **sys** provides system-level functionality. **sys.argv** is the list of command line arguments starting from the name of the Python program file.

|  |
| --- |
| **import** sys  **for** arg **in** sys.argv:  print arg |

**Assignments:**

* Run the code above with different parameters.
* Write a Python program that prints itself.

This code prints all parts of the command line. Note that all values (elements of **sys.argv**) are strings. To pass a numerical value you need to convert the corresponding parameter:

|  |
| --- |
| **import** sys  x = sys.argv[1]  print x, type(x)  y = int(x)  print y, type(y) |

Function **type** returns type of the argument: string, integer, float, list, etc.

## File input/output, parsing strings

Depending on the file format, parsing its content may be more or less difficult. For purposes of the course we need to parse maze descriptions saved in text files and convert them into trees.

Here is a Python program that reads and prints the content of the file:

|  |
| --- |
| **import** sys  f = open(sys.argv[1],"r")  file\_content = f.read()  f.close()  print file\_content |

First we need to split the file content into lines. Lines in files end with an invisible end-of-line (EOL) symbol or two symbols depending on the operating system convention. In most cases the symbol is “new line” (the special symbol printed as \n)or two symbols “carriage return, new line” (\r\n). A backslash preceding a letter indicates that the letter is being interpreted differently than if it were plain text. The following code splits file content into lines:

|  |
| --- |
| lines = file\_content.split("\n") |

**Assignment:** Combine two pieces of code and confirm the type of variable **lines** (List). Print it element by element.

The next step is to convert each line into a list of logical constants (True, False) so that True corresponds to a path and False corresponds to a wall. For example, convert **0101010** into [ False,True,False,True,False,True,False]. One way of doing this is to go character by character in a loop over the entire string, compare characters with 1 and 0, and append the new logical value to a list:

|  |
| --- |
| line = "01010110"  num = len(line)  L = []  for i in range(num):  is\_path = line[i] == "1"  L.append(is\_path) |

In this piece of code, we first use function **len** to calculate number of characters in the line. Next, we create an empty list **L**. We go character by character using the character’s index in the string and comparing the character with “1”. Logical variable **is path** is initialized depending on the result of comparison with “1” and then appended to the list.

This way of coding is valid, but not very efficient or compact. Instead of using a loop we can use so-called list comprehensions. For a list **L** and a function **f** the following expression

**[f(x) for x in L]**

creates a list constructed by applying function **f** to each element of the list. For example:

|  |
| --- |
| def f1(x):  return 2\*x  def f2(x):  return x\*x  L = [1,2,3,4]  L1 = [f1(x) for x in L]  L2 = [f2(x) for x in L]  print L, L1, L2 |

or the same in place (without defining functions):

|  |
| --- |
| L = [1,2,3,4]  L1 = [2\*x for x in L]  L2 = [x\*x for x in L]  print L, L1, L2 |

A string in Python can be treated as a list (with some limitations). The following code converts string **line** into a list of characters it contains:

|  |
| --- |
| line = "abc"  L = [c for c in line]  print line, L |

Now we can rewrite the code converting strings element by element into logical variables in the following way:

|  |
| --- |
| line = "01010110"  L = [x=="1" for x in line] |

Here we use the in-place expression **x==“1”** for each element of the **line**.

## Exercises for A.7-A.8

1. Write program that prints sum of all command line arguments. Write the following versions:
   1. Assume all arguments are numbers. Use a loop.
   2. Assume all arguments are numbers. Use list comprehensions.
   3. Assume all arguments are strings. Join them in a loop and using **join** function.
2. Create a text file with a column of numbers. Write program that reads this file and prints sum of numbers, max value, min value, indices (line numbers) of max and min values.
3. Create a comma-separated file with numbers. Write program that reads this files and prints sum of elements in i-th columns, j-th row (i and j are parameters). Write versions using loops and list comprehensions.

## Image processing

To proceed you need to install Python module Pillow. Run

**pip install Pillow**

or refer to the online documentation (python-pillow.org) for your operating system.

Here is the Python code that loads images from PNG files, prints an image’s size and mode, converts the image into a list of pixels, prints the maze definition, creates a new list of RGB pixels, changes color of several pixels to red, and creates an image for the new list and saves it.

|  |
| --- |
| **from** PIL **import** Image  img = Image.open("maze1s.png")  img\_list = list(img.getdata())  w, h = img.size  **for** i **in** range(h):  row = ["0" **if** p>192 **else** "1" \  **for** p **in** img\_list[(i\*w):((i+1)\*w)]]  print "".join(row)  new\_img\_list = [(x,x,x) **for** x **in** img\_list]  **for** i **in** range(100):  new\_img\_list[i\*w+i] = (255,0,0)  new\_img = Image.new( "RGB", img.size)  new\_img.putdata(new\_img\_list)  new\_img.save("test.jpg") |

Let’s review the code line by line:

* + - Import support for images from the PIL package.
    - Load image from the PNG file.
    - Extract pixel data and convert it into a list of pixel colors.
    - Extract image size: width and height.
    - In the loop over each row, we create and print the maze description (“0” for wall, “1” for an element of the path). The original image is in grayscale: shades of gray are coded as integers between 0 and 255. High values are lighter, low values are darker. Everything above 192 is considered to be a wall; everything darker is a path.
    - Create a new list: RGB triplets representing the same colors as in the original image (all three components—red, green, blue—are identical).
    - To illustrate pixel color operations, we change 100 pixels on the diagonal, making them red. RGB value (255,0,0) corresponds to red color.
    - Create a new image object in RGB color mode and the same size as in the original image.
    - Create image pixel data from the list of RBG triplets.
    - Save the new image as a JPEG file.

You can use basic image operations to illustrate differences between graph search algorithms.

## Exercises for A.9

You can use any image file: picture from your phone, image downloaded from the web, image created with a graphical editor.

1. Create 10x10 RGB image file: blue background, red boundaries, green diagonals.
2. Crop an image: return rectangular area given by top-left and bottom-right corner.
3. Flip an image horizontally, vertically.

## Python modules

Modules in Python are the way to organize and reuse code. In the simplest case, a module is just a Python file with functions or other Python elements.

If file **A.py** contains the following code:

|  |
| --- |
| **def** f(x):  print "Function from A.py"  **return** x\*x |

it can be used from other Python programs or in the Python interactive shell:

|  |
| --- |
| **import** A  print A.f(2) |

or

|  |
| --- |
| from A import f  print f(2) |

Bigger projects may include multiple files. In this case, the module is organized by directory. All files are saved to the directory, and the name of the directory becomes the name of the module. In addition to the Python files with the module-specific functions, one more service file is needed: \_\_**init\_\_.py**. The file may be empty, but normally it contains the list of elements.

Consider the following configuration: directory **module1** containing three files: **func1.py, func2.py, \_\_init\_\_.py**. The code in the files looks like this:

|  |
| --- |
| # file \_\_init\_\_.py  \_\_all\_\_= ["func1","func2"] |

|  |
| --- |
| # file func1.py  **def** f1(x):  print "Function f1 from func1 from module1"  **return** x\*x  **def** f2(x):  print "Function f2 from func1 from module1"  **return** -x |

|  |
| --- |
| # file func2.py  **def** f1(x):  print "Function f1 from func2 from module1"  **return** x\*x\*x  **def** g(x):  print "Function g from func2 from module1"  **return** x + 1000 |

There are several ways to access functions from this module:

|  |
| --- |
| **import** module1.func1  **import** module1.func2  print module1.func1.f1(2)  print module1.func1.f2(7)  print module1.func2.f1(3)  print module1.func2.g(234) |

|  |
| --- |
| **from** module1 **import** \*  print func1.f1(2)  print func1.f2(7)  print func2.f1(3)  print func2.g(234) |

|  |
| --- |
| **from** module1.func1 **import** \*  **from** module1.func2 **import** f1 **as** f1A  **from** module1.func2 **import** g  print f1(2)  print f2(7)  print f1A(3)  print g(234) |

## Random numbers

There exist algorithms that require a random number generator. You may want to simulate a coin toss and generate a sequence of 0s and 1s, or to generate a random initial state, as we did in the section of this course dealing with the 8-puzzle.

Python provides support for random numbers in the **random** module. Here is the code that can be used to generate a sequence of 0s and 1s:

|  |
| --- |
| **from** random **import** randint  **for** \_ **in** range(10):  print randint(0,1) |

The function **randint(A,B)** from the module **random** returns a random integer from A to B inclusive. If you call this function many times, each number will appear an equal number of times: The function generates numbers that are **uniformly distributed**. You can adjust the code above and confirm this; just count the number of 0s and 1s after 100 or 1000 calls of the function.

The other useful function from the module **random** is **random** (the function name is the same as the module name):

|  |
| --- |
| **from** random **import** random  **for** \_ **in** range(10):  print random() |

The function doesn’t take a parameter and returns a number between 0 and 1. Numbers are also uniformly distributed: If you call this function many times, the number of cases when the function returns a value in any interval of a given size will be the same. For example, the number of cases when the value is below 0.5 and above 0.5 (intervals (0,0.5) and (0.5,1) are of equal sizes), and likewise in the interval (0.23,0.33) and (0.58,0.68).

For purposes of this course we don’t need other types of distributions or other functions from the module.

## Exercises for A.11

1. Generate N (N is a parameter) random numbers uniformly distributed in (0,1). Count the number of cases in an arbitrary interval given by left and right boundaries, for example in (0.1,0.2).
2. Simulate coin tossing. For N outcomes find the length of the longest sequence of heads or tails. For example, for HHTHTTTHHTHTTH return 3 for T.

## Classes

Object-oriented programming (OOP) is one of the most powerful and widely used programming concepts. Python supports OOP and its core element: the notion of classes. For purposes of this course we’ll consider just basic ideas without covering all aspects of the concept. [[10]](#footnote-10)

The idea of a class is to keep data and related functionality together. Your program may work with objects described by object-specific data. For example, geometrical figures, bank accounts, or game boards. Geometrical figures may be given by a center and radius (circle) or by several sizes (triangle, rectangle) or by positions of corners (arbitrary polygon). A bank account keeps information about the owner (name, address, email, phone number), balance, transactions, etc.

When developing a program that works with multiple geometrical figures, we may need each figure to have a common interface functions: for example, the function that returns a position of a geometrical center, or a function that returns True if a point is inside the figure and False otherwise. These functions are different for different geometrical figures, and it’s convenient to isolate this functionality; the program only “knows” that each geometrical figure has functions **center()** and **is inside(point)**.

To do this we first declare a base class common for all figures and implementing common functionality:

|  |
| --- |
| **class** figure(object):  **def** \_\_init\_\_(self,name):  self.name = name  **def** center(self):  **pass**  **def** is\_inside(self,point):  **pass**  **def** \_\_repr\_\_(self):  **return** self.name |

Let’s go line by line:

#### class figure(object):

We’ve declared a new class, **figure,** extending the base class **object**. **Object** is the base class for almost everything in Python: integers, lists, strings, etc., are all objects. What this means is that each derived class keeps (inherits) the functionality implemented in the base class.

#### def \_\_init\_\_(self,name):

Function **init** is called each time we create a new object. All member functions[[11]](#footnote-11) have the first parameter **self**: the reference to the constructed object. \_\_i**nit\_\_**may have other parameters to initialize member variables. Here the parameter is **name**: the name of the figure. The base class doesn’t know anything else; all other details will be implemented in classes derived from **figure**.

#### self.name = name

This line initializes member variable **self.name**. If we have an instance of this class **f** the variable **name** can be accessed as **f.name**. Inside the class definition the first part of this expression (the reference to the object) is **self**,and the variable is accessed as **self.name**.

Next, we declare two functions: **center** and **is inside**. We are using the **pass** statement to do nothing in the base class.

#### def \_\_repr\_\_(self ):

Function **repr** returns a string representation of the object. It’s called each time we print something; so, Python statement **print x** prints the string returned by **x.\_\_repr\_\_()**. For the base class this function returns the name of the figure.

Now we want to implement a rectangle. For demonstration purposes we’ll assume the corners at points (0*,* 0)*,* (*w,* 0)*,* (0*, h*)*,* (*w, h*). We need only two numbers to describe any rectangle: width **w** and height **h**.

|  |
| --- |
| **class** rectangle(figure):  **def** \_\_init\_\_(self,w,h):  figure.\_\_init\_\_(self, "rectangle")  self.w = w  self.h = h  **def** center(self):  **return** (self.w/2,self.h/2)  **def** is\_inside(self,point):  **return** point[0]>0 **and** point[0]<self.w **and** \  point[1]>0 **and** point[1]<self.h  **def** \_\_repr\_\_(self):  **return** "{}, width={}, height={}" \  .format(self.name,self.w,self.h) |

Note that the \_\_**init\_\_** function now has different parameters. In the beginning it calls the corresponding function of the base class **figure**. In this case the initialization is trivial, but in more complicated cases the reuse may be extremely helpful. **\_\_repr\_\_** returns the string with name and parameters.

Here is the class implementing the same functions for a circle given by radius **r** and the position of center **c**:

|  |
| --- |
| **class** circle(figure):  **def** \_\_init\_\_(self,r,c):  figure.\_\_init\_\_(self, "circle")  self.r = r  self.c = c  **def** center(self):  **return** self.c  **def** is\_inside(self,point):  **return** (point[0]-self.c[0])\*\*2 + (point[1]-self.c[1])\*\*2 \  < self.r\*\*2  **def** \_\_repr\_\_(self):  **return** "{}, radius={}, center={}" \  .format(self.name,self.r,self.c) |

Now let’s learn how to use these classes:

|  |
| --- |
| fig = figure("unknown figure" )  rect = rectangle(2.0,1.0)  circ = circle(3.0,(1.0,4.0))  print type(fig), fig  print fig.center(), fig.is\_inside((0.0,0.0))  print type(rect),rect  print rect.center(), rect.is\_inside((0.5,1.2))  print type(circ), circ  print circ.center(), circ.is\_inside((2.0,4.0)) |

We first created three objects: generic figure, rectangle and circle. Then we called their corresponding functions. The same can be done using a loop:

|  |
| --- |
| L = []  L.append( figure("unknown figure" ) )  L.append( rectangle(2.0,1.0) )  L.append( circle(3.0,(1.0,4.0)) )  **for** f **in** L:  print type(f), f  print f.center(), f.is\_inside((0.5,1.0)) |

This section covers only the basics of Python classes, the minimum needed for this course.

## Saving Python structures. JSON.

Sometimes you need to save results of calculations for future use. Sometimes the task is simple: A number can be saved in a text file, and a list of numbers may be a column in a comma-separated (CSV) file. What if we need to save a more complex, nested structure? Consider the following variable:

|  |
| --- |
| X = {}  X["abc"] = [ 1, 2, [ 3, [4, 5, 6] ], 7, 8 ]  X["def"] = { "A":34.567, "B":98.765 }  X["ghi"] = "Sample text" |

Each element value of this dictionary has a different type: nested list, dictionary, and string. We can introduce a custom solution for this particular case, but it may not work for other cases. It would be convenient to have a way to save arbitrary Python variables and then load them without doing task-specific parsing.

The Python module **json [[12]](#footnote-12)** allows to do exactly this. Here is how we can save the variable **X** from the example above. We import a new module, create a variable, open a file for writing and dump the variable to the file using the **json.dump** function.

|  |
| --- |
| **import** json  X = {}  X["abc"] = [ 1, 2, [ 3, [4, 5, 6] ], 7, 8 ]  X["def"] = { "A":34.567, "B":98.765 }  X["ghi"] = "Sample text"  f = open("example.json","w")  json.dump(X,f)  f.close() |

Run this code and make sure the file is created. Now we can load the variable:

|  |
| --- |
| **import** json  f = open("example.json","r")  Y = json.load(,f)  f.close()  print Y |

Try this and confirm that the variable **Y** corresponds to **X** from the previous example.

## Python: Machine learning libraries

This course includes a chapter on artificial neural networks (NN). There are several Python libraries that provide tools to configure, train, test, and use NN: scikit-learn, Tensorflow, pybrain, and Caffee, among others. For purposes of the course we’ll be using **scikit-learn**. Other libraries and stand-alone applications are used in many different projects and have important features, but **scikit-learn** covers a very wide range of techniques, and it’s important to learn how to use it. **Scikit-learn** is not the fastest solution, but it’s very convenient for prototyping and testing ideas. One indirect way of using the library is to review its documentation; the table of contents can serve as a reference to many techniques of various levels of complexity.

Please refer to the online documentation[[13]](#footnote-13) to install the library. **scikit-learn** depends on two other libraries—**numpy** and **scipy**—they’ll be installed as well.

A **numpy** library provides high-performance implementation of arrays and related functionality. In this course we’ll be using **numpy** arrays together with Python lists. A **numpy** array can be constructed from a list and vice versa:

|  |
| --- |
| **import** json  # Python list  L = [1,2,3,4]  print type(L), L  # numpy array from the list  A = np.array(L)  print type(A), A  # Python list from numpy array  L2 = A.to\_list()  print type(L2), L2 |

The constructor **np.array** takes as a parameter a Python list. **Array** member function **tolist()** returns the list.

**scikit-learn** provides a lot of functionality, and we won’t cover all use cases, but the basic work flow in the case we have an input **X** and associated output **Y** and want to build a model that can derive output based on provided input includes several steps:

* + 1. Initialize a model: Choose the type of the model and parameters.
    2. Train the model (fit): Provide the model with inputs and outputs and run the optimization to reduce the error.
    3. Use the model to check the accuracy for data used in training (in-sample) and for data not use in training (out-of-sample).

Here is an example of using **scikit-learn** to train a multi-layer perceptron and get results:

|  |
| --- |
| **import** numpy **as** np  **from** sklearn.neural\_network **import** MLPRegressor  # ...  # we are assuming that trainig data are loaded as Python lists:  # X\_input and Y\_output  # ...  X = np.array(X\_input)  Y = np.array(Y\_output)  nn = MLPRegressor(hidden\_layer\_sizes=(9))  n = nn.fit(X,Y)  Y\_predicted = nn.predict(X) |

**MLPRegressor** is one of the models implemented in **scikit-learn**. In this example we are creating an object **nn**: a perceptron with one hidden layer that contains 9 neurons. For all other parameters—type of activation function, optimization algorithm—the model takes default values[[14]](#footnote-14). In the code above, we are creating inputs and outputs as **numpy** arrays, initializing **MLPRegressor**, and fitting the model. **Y predicted** contains results of applying the trained model to inputs **X**. **Y\_predicted** can be compared with **Y** to evaluate the accuracy.

For assignments in this course we’ll need **MLPRegressor** initialized with more parameters:

|  |
| --- |
| ...  nn = MLPRegressor(hidden\_layer\_sizes=(9),  activation = "logistic",  solver = "lbfgs") |

Activation **‘logistic’** refers to *f* (*x*) = 1*/*(1 + *exp*( *x*)) (see the section on neural networks for details). Solver **‘lbfgs’** is an optimization algorithm[[15]](#footnote-15). This course doesn’t cover details of optimization algorithms.

*−*

## Coding problems

The best way to build practical skills in programming is to do real development and solve practical problems with your language of choice. In this section we present coding problems, from elementary to more complex. You can do them in an arbitrary order. Those of you who are interested in solving advanced problems can start with Project Euler [[16]](#footnote-16).

Problems (try to find more than one solution):

* + 1. Write a function that takes a day of a week (for example, “Monday”) as a parameter and returns True for weekends and False for weekdays.
    2. Write a function that takes a day of a year and returns the month. For example, given 67 it returns “March”.
    3. Define night as the time between 22:00 and 5:50. Write a function that takes hours and minutes and returns True if it’s night and False otherwise. For example, the function will return True for 23 and 4:41, and False for 5:51.
    4. Write a function that parses string with a date and returns all elements—year, month, day—as integers. For example, for “January 04, 2018” it returns (2018,1,4). You can assume this format of the date (month as a string, two-digit day, comma, four-digit year).
    5. Write a function that removes all spaces from a given string and capitalizes the first letter of every word. For example it converts “This is a sample string.” into “ThisIsASampleString.”
    6. Write a function that reads a text file and returns word frequencies: For each word it calculates the number of cases where it appears in the text.
    7. Write a function that returns all unique elements of a list. For example, for [1,2,1,4,5,2] it returns [1,2,4,5].
    8. Write a function that returns the sum of each of two sequential elements of a list. For example, for [1,2,3,10,20] it returns [3,5,13,30].
    9. Write a function that sorts a list (do not use the sorting functions provided by Python).
    10. Write a function that for a given dictionary (key:value pairs) returns (value:[key]) dictionary. For example, for {“a”:1, “b”:2, “c”:1} it returns {1:[“a”,“c”], 2:[“b”]}
    11. Write a function that rounds a number (do not use the rounding functions provided by Python).
    12. Write a function that transposes a list of lists. For example, for [[1,2,3],[4,5,6]] it returns [[1,4],[2,5],[3,6]].
    13. Write a function that returns True for prime numbers and False otherwise.
    14. Write a function that for a given number returns all smaller prime numbers. For example, for 15 returns [2,3,5,7,11,13].
    15. Write a function that returns all possible combinations of two elements for a given list. For example, for [1,2,3] it returns [[1,2],[1,3],[2,3],[2,1],[3,1],[3,2]] (do not use functions available in Python).

1. More elegant and efficient implementation of trees requires deeper programming skills. [↑](#footnote-ref-1)
2. You can rewrite the code using a more efficient Python structure: **dqueue**. [↑](#footnote-ref-2)
3. We’ll assume that all weights are positive [↑](#footnote-ref-3)
4. The question of convergence of this iterative procedure is not covered in this course. [↑](#footnote-ref-4)
5. Function **random()** returns values evenly distributed between 0 and 1. Thus, the value below 0.2 appears on average one time out of five. [↑](#footnote-ref-5)
6. 1The biological principles are usually seen in AI as metaphors rather than a theoretical foundation. [↑](#footnote-ref-6)
7. It’s common to use 0 to denote False and 1 to denote True [↑](#footnote-ref-7)
8. At this moment we do not consider the question of using NN in- and out-of-sample. [↑](#footnote-ref-8)
9. 4**itertools** is a part of Python and doesn’t require installation. [↑](#footnote-ref-9)
10. If you are familiar with OOP in other languages (e.g., C++) you’ll find that implementation of classes in Python assumes members (variables and functions) are public, and functions are virtual. [↑](#footnote-ref-10)
11. 3Except static functions, which are not covered here. [↑](#footnote-ref-11)
12. JSON stands for JavaScript Object Notation. This format is widely used for data exchange. [↑](#footnote-ref-12)
13. scikit-learn.org [↑](#footnote-ref-13)
14. Please review online documentation for details [↑](#footnote-ref-14)
15. 7LBFGS: Limited-memory Broyden-Fletcher-Goldfarb-Shanno [↑](#footnote-ref-15)
16. 8https://projecteuler.net/ [↑](#footnote-ref-16)