

Consider the following 3×3 grid.

	col 0	col 1	col 2
row 0	0	1	2
row 1	3	4	5
row 2	6	7	8

Remark 0.1. Note that in this exercise, indexing always starts from 0.

Exercise 1

The row of an index i in an $m \times n$ is given by the floor division $i//n$, and the column of i is given by the modulus $i\%n$. For a 3×3 matrix, we have $n = 3$. So for an index i , the quotient $i//3$ gives the row of i , and $i\%3$ gives the column of i .

The functions `row(i)` and `col(i)` take as argument an integer i between 0 and 8 inclusive and return the quotient $i//3$ representing the row of i and the remainder $i\%3$ representing the column of i respectively.

For example, index 5 is in row $5//3 = 1$, which is the second row. It is also in col $5\%3 = 2$, which is the third column.

```

1  def row(i : int)->int:
2      """
3      Determine the row of index 'i' in a 3x3 grid.
4
5      Parameters
6      -----
7      i : int
8          An index between 0 and 8.
9
10     Returns
11     -----
12     int
13         The row number (0, 1, or 2) of the index 'i'.
14
15     Example
16     -----
17     row(5) -> 1 (Second row)
18
19     Raises
20     -----
21     TypeError
22         If 'i' is not an integer.
23     IndexError
24         If 'i' is not between 0 and 8 inclusive.
```

```

25     """
26
27     if not isinstance(i, int):
28         raise TypeError(f"Sorry. '{i}' must be an integer")
29
30     if i not in range(0, 9):
31         raise IndexError(f"Sorry. '{i}' must be a number between 0 and 8
inclusive.")
32
33
34     return i//3
35
36 def col(i:int) -> int:
37     """
38     Determine the column of index 'i' in a 3x3 grid.
39
40     Parameters
41     -----
42     i : int
43         An index between 0 and 8.
44
45     Returns
46     -----
47     int
48         The column number (0,1, or 2) of the index 'i'.
49
50     Example
51     -----
52     col(5) -> 2 (Third column)
53
54     Raises
55     -----
56     TypeError
57         If 'i' is not an integer.
58     IndexError
59         If 'i' is not between 0 and 8 inclusive.
60     """
61     if not isinstance(i, int):
62         raise TypeError(f"Sorry. '{i}' must be an integer")
63
64     if i not in range(0, 9):
65         raise IndexError(f"Sorry. '{i}' must be a number between 0 and 8
inclusive.")
66
67
68     return i%3

```

Listing 1: Exercise 1

Exercise 2

The function `intermediate_node(i : int, j : int)` aims to determine whether there is an intermediate node between two given nodes i and j in a 3×3 grid, with nodes numbered 0 through 8, and if so, returns the intermediate node.

The function begins by initializing two variables `col_set` and `row_set`. The variable `col_set` is assigned the set of column indices $\{col(i), col(j)\}$ of nodes i and j . The variable `row_set` is assigned the set $\{row(i), row(j)\}$ of row indices of the nodes i and j .

1. To ensure i and j are in different rows, we need $abs(row(i) - row(j)) == 2$. This implies the length of `row_set` is 2.
2. To ensure i and j are in different columns, we need $abs(col(i) - col(j)) == 2$. This implies the length of `col_set` is 2.
3. To ensure i and j are in the same rows, the length of `row_set` must be 1.
4. To ensure i and j are in the same columns, the length of `col_set` must be 1.

Now, if $abs(row(i) - row(j)) == 2$ and $len(col_set) == 1$, then i and j are in the same column with an intermediate node $\frac{i+j}{2}$ between them.

Next, if $abs(col(i) - col(j)) == 2$ and $len(row_set) == 1$, then i and j are in the same row with an intermediate node $\frac{i+j}{2}$ between them.

Finally, if $len(row_set) == 2$, $len(col_set) == 2$, $abs(row(i) - row(j)) == 2$ and $abs(col(i) - col(j)) == 2$ the i and j lie in a diagonal, with an intermediate node $\frac{i+j}{2}$ between them.

In summary, three sets of conditions are tested. If each set is satisfied, an intermediate node is returned by the function. Otherwise, it returns -1 .

```

1  def intermediate_node(i: int, j: int) -> int:
2      """
3          Determine the intermediate node 'k' lying between 'i' and 'j' in a 3
4          x3 grid, if it exists.
5
6          Parameters
7          -----
8          i : int
9              An index between 0 and 8.
10         j : int
11             An index between 0 and 8.
12
13         Returns
14         -----
15         int

```

```

16         The node 'k' lying between 'i' and 'j' if it exists, otherwise
17         -1.
18
19     Raises
20     -----
21     TypeError
22         If 'i' or 'j' is not an integer.
23     IndexError
24         If 'i' or 'j' is not between 0 and 8 inclusive.
25
26     Examples
27     -----
28     >>> intermediate_node(0, 8)
29     4
30     >>> intermediate_node(1, 5)
31     -1
32     """
33     if not all(isinstance(index, int) for index in (i, j)):
34         raise TypeError(f"Sorry, both '{i}' and '{j}' must be integers")
35
36     if not all(0 <= index <= 8 for index in (i, j)):
37         raise IndexError(f"Sorry, both '{i}' and '{j}' must be numbers
38         between 0 and 8 inclusive.")
39
40     col_set = {col(i), col(j)}
41     row_set = {row(i), row(j)}
42
43     # Checking for an intermediate in a column.
44     if abs(row(i) - row(j)) == 2 and len(col_set) == 1:
45         return (i + j) // 2
46
47     # Checking for an intermediate in a row.
48     if abs(col(i) - col(j)) == 2 and len(row_set) == 1:
49         return (i + j) // 2
50
51     # Checking for an intermediate in a diagonal.
52     if len(row_set) == 2 and len(col_set) == 2 \
53         and abs(row(i) - row(j)) == 2 and abs(col(i) - col(j)) == 2:
54         return (i + j) // 2
55
56     return -1

```

Listing 2: Exercise 2

Exercise 3

In this exercise, the function `is_admissible(pattern, i)` is defined. It takes two arguments, `pattern` and an index `i`, and determines if `pattern` can be extended to `pattern + (i,)`.

The function starts by checking if `i` belongs to `pattern`; it returns false if this is the case. This ensures that a node is not visited more than once.

Next, `pattern` is extended to a new pattern `pattern + (i,)`. The extension is assigned to a variable called `new_pattern`. This new pattern is passed as an argument to a helper function `is_valid_pattern(new_pattern)` which returns true or false depending on the validity of `new_pattern`. If true is returned, then `pattern` can be extended to `pattern + (i,)`. Otherwise, it cannot be extended.

The helper function, `is_valid_pattern(pattern : tuple)`, takes as argument a tuple of integers ranging from 0 through 8 and returns true or false depending on the validity of pattern. A pattern `p` is deemed valid, if it satisfies the following:

1. `p` is valid if it is the empty tuple `()`.
2. If `p` is not empty then
 - (a) loop through its elements to ensure a node is not repeated. If a node appears more than once, then `p` is not a valid pattern; the function returns false.
 - (b) if an intermediate `k` exists between two consecutive nodes `i` and `j` in `p` and `(i, k, j)` is not a subset of `p`, then `p` is not a valid pattern. Again the function returns false.

The function returns true otherwise.

```
1  def is_admissible(pattern: tuple, i: int) -> bool:
2      """
3          Determines if the index i can be appended to the given pattern
4
5          Parameters
6          -----
7          pattern: tuple of int
8              A tuple of digits from 0 to 8 representing a valid swipe pattern
9          i: int
10             An integer between 0 to 8 representing a potential extension
11 node.
12
13 Returns
14 -----
15 bool
16     True if index i can be appended to pattern, False otherwise.
17     """
```

```

18         if i in pattern:
19             return False
20
21         new_pattern = pattern + (i,)
22         return is_valid_pattern(new_pattern)
23     #####
24     # Helper function for Exercise 3: is_valid_pattern.
25     #####
26
27     def is_valid_pattern(pattern):
28         """
29         Determines if the given swipe pattern is valid according to Android
30         pattern rules.
31
32         Parameters
33         -----
34         pattern : tuple
35             A tuple of digits from 0 to 8 representing a swipe pattern.
36
37         Returns
38         -----
39         bool
40             True if the pattern is valid, False otherwise.
41         """
42
43         # Empty pattern ()
44         if len(pattern) == 0:
45             return True
46
47         # Non-empty pattern
48         for i in range(1, len(pattern)):
49             if pattern[i] in pattern[:i]:
50                 False
51
52             intermediate = intermediate_node(pattern[i-1], pattern[i])
53             if intermediate != -1 and intermediate not in pattern[i-1:i+1]:
54                 return False
55         return True
56     #####

```

Listing 3: Exercise 3

Exercise 4

In this exercise, the function *extensions*(*pattern* : *tuple*) takes as argument a valid pattern represented as a Python tuple, and returns the set of all admissible extensions of this pattern by

1. Thus, *exten(pattern : tuple)* will rely on the function *is_admissible(pattern : tuple, i : int)* to generate this set of admissible extensions.

Here, a for loop comes in handy. We loop through the nodes from 0 through 8 and ckeck if they are admissible to the pattern. An empty set, *set_of_tuples = set()*, is initialized. If *i* is one of the numbers from 0 through 8, and *is_admissible(pattern, i)* returns true, then *pattern + (i,)* is added to the set *set_of_tuples*. This repeats until the for loop ends.

Finally, the function *extensions(pattern)* returns the set *set_of_tuples*.

```
1  def extensions(pattern: tuple) -> set:
2      """
3      Extends a pattern by one more node.
4
5      Parameters
6      -----
7      pattern : tuple
8          A tuple of digits from 0 to 8 representing a valid swipe
9          pattern
10
11     Returns
12     -----
13     The set of all possible extensions of pattern by 1 more node.
14     """
15
16     set_of_tuples = set()
17
18     for i in range(9):
19         if is_admissible(pattern, i):
20             extended_pattern = pattern + (i,)
21             set_of_tuples.add(extended_pattern)
22
23     return set_of_tuples
```

Listing 4: Exercise 4

Question: What would go wrong here if we instead chose to represent our patterns as lists rather than tuples? How could you modify your approach to accommodate this?

If we choose to represent our patterns as lists, rather than tuples, then

- the expression *pattern + (i,)* will throw a `TypeError` exception. This is because a list and a tuple cannot be concatenated. If a list is chosen, then we need to first convert the list to a tuple, concatenate.
- Also, the function *is_admissible* will throw a `TypeError` exception since it takes as argument a tuple of integers.

```
1  def extensions(pattern: list) -> set:
```

```

2      '''
3      Parameters
4      -----
5      pattern : list
6          A list of digits from 0 to 8 representing a valid swipe
7          pattern
8
9      Returns
10     -----
11         The set of all possible extensions of pattern by 1 more
12         node.
13     '''
14
15     set_of_tuples = set()
16
17     for i in range(9):
18         if is_admissible(tuple(pattern), i):
19             extend_pattern = tuple(pattern) + (i,)
20             set_of_tuples.add(extend_pattern)
21
22     return set_of_tuples

```

Listing 5: Modification for Exercise 4

Exercise 5

In this exercise, we write a Python function *generate_patterns()* which outputs a Python dictionary whose keys are integers 0 through 9 and whose value at integer i is the set of all swipe patterns of length i .

The function begins by first creating a dictionary whose keys are integers from 0 through 9 and whose initial values at each integer is an empty set. The first for loop is used to create a set of patterns of length one. So, the patterns of the form $(i,)$ are added to the set *pattern*[1], where i is an integer from 0 through 8. The next for loop is used to create the other patterns of length 2 through 9.

Next, two nested for loops are used to create the other patterns of length 2 through 9. The outer for loop loops through the keys k of the dictionary starting from 2 through 9. The inner for loop loops through the set *patterns*[$k - 1$] and each pattern of length $k - 1$ is passed to the function *extensions* from Exercise 5 to generate a set of admissible patterns of length k . The set *patterns*[k] is then updated with the set of admissible patterns.

The function *generate_patterns()* then returns the dictionary *pattern*.

```

1      def generate_patterns():
2          '''
3          Returns

```



```
4      -----
5      A dictionary whose keys are integers 0 through 8 and whose
6      value at integer i is the set of all swipe patterns of length i
7      '''
8
9      patterns = {k : set() for k in range(10)}
10
11     # patterns of length 1
12     for i in range(9):
13         patterns[i].add((i,))
14
15     for k in range(2,10):
16         for pattern in patterns[k-1]:
17             new_pattern = extensions(pattern)
18             patterns[k].update(new_pattern)
19     return patterns
```

Listing 6: Sample Python Code

Exercise 6

How many possible swipe patterns are there of length 4 or more? How many patterns of length 9?

Generate all patterns with the function *generate_patterns* from Exercise 5. Use tuple comprehension – *sum(len(pattern_dict[i]) for i in range(4, 10))* – to sum all the patterns of length 4 or more. Here *len(pattern_dict[i])* computes the length of patterns in the set *pattern_dict[i]* of length *i*. The for loop loops over the number from 4 through 9.

Also, patterns of length 9 is *len(pattern_dict[9])*. The results are printed using f-string notation.

```
1     pattern_dict = generate_patterns()
2     pattern_length_4_or_more = sum(len(pattern_dict[i]) for i in range(4,10)
3     )
4     pattern_length_9 = len(pattern_dict[9])
5     print(pattern_length_9)
6     print(f"Number of possible swipe patterns of length 4 or more: {
7     pattern_length_4_or_more}")
8     print(f"Number of possible swipe patterns of length 9: {pattern_length_9
9     }")
```

Listing 7: Exercise 6

Output of code Snippet

- Number of possible swipe patterns of length 4 or more: 139880

- Number of possible swipe patterns of length 9: 32256
-

Exercise 7

I watched my friend type in her password over her shoulder. I saw that her pattern has the following properties:

- It starts by connecting node 1 to node 0
- It uses every node
- The final edge connected node 7 to node 8
- At some point, I saw her connect node 4 to node 6

How many possible patterns fit this description?

The code snippet below filters the patterns that fit the description above. Here, an empty list [] is assigned to the variable *filtered_patterns*. Since every node will be used, then all patterns that fit the description are of length 9. A for loop is used to loop through all the patterns of length 9 in the set *pattern_dict*. The if statement ensure that the pattern first connects node 1 to 0. It also ensures that the last ends with node 7 connected to 8. It also ensures that node 4 and 6 are in the pattern, and that node 4 comes before 6. The patterns that fit the condition are appended to *filtered_patterns* and its length computed.

```
1 filtered_patterns = []
2 pattern_dict = generate_patterns()
3 for pattern in pattern_dict[9]:
4     if (pattern[0], pattern[1]) == (1,0) and pattern[-2:] == (7,8)\
5         and 4 in pattern and 6 in pattern\
6         and 4 in pattern[:pattern.index(6)+1]:
7         filtered_patterns.append(pattern)
8 number_of_valid_patterns = len(filtered_patterns)
```

Listing 8: Exercise 7

Output of code Snippet

- Number of possible swipe patterns fitting the description is: 18
-

Optional Task

In this section we loop at some metrics for measuring pattern complexity.

Metrics and Code snippets

The following are some metrics for measuring the complexity of a pattern.

- **Length of a pattern**
- **Change in direction:** Counting changes in direction between consecutive moves
- **Change in angle:** Counts the number of angle changes (e.g., straight, right angles, obtuse angles) in the pattern

The following code snippets implements the metrics given above

```
1  import android_swipe_pattern as asp
2  from numpy import array, sqrt, dot, degrees, arccos
3
4  def compute_angle(x: tuple[int, int], y: tuple[int, int], z: tuple[int,
5  int]) -> float:
6      """
7      Calculate the angle between three points.
8
9      Parameters
10     -----
11     x, y, z : tuple[int, int]
12         Coordinates of the points.
13
14     Returns
15     -----
16     float
17         The angle in degrees between two vectors fromed by the three
18     points.
19     """
20
21     def vector(a,b):
22         return (b[0]-a[0], b[1]- a[1])
23     # Angle formula:  $v \cdot u = |u| |v| \cos(\theta)$ 
24
25     u = vector(x,y)
26     v = vector(z,y)
27
28     dot_product = dot(array(u), array(v))
29     norm_u = sqrt(u[0]**2 + u[1]**2)
30     norm_v = sqrt(v[0]**2 + v[1]**2)
```

```

29     cos_theta = dot_product/ (norm_u * norm_v)
30
31     return degrees(arccos(cos_theta))

```

Listing 9: Code snippet to compute angle between two vectors

```

1  def angle_change(pattern: tuple[int, ...]) -> int:
2      """
3      Compute the number of angle changes in the pattern.
4
5      Parameters
6      -----
7      pattern: tuple[int, ...]
8          A tuple of digits from 0 to 8 representing a swipe pattern.
9
10     Returns
11     -----
12     int
13         The number of significant angle changes in the pattern.
14     """
15     coordinate_of_nodes = [(asp.row(node), asp.col(node)) for node in
pattern]
16     number_of_angle_changes = 0
17
18     for i in range(1, len(coordinate_of_nodes)-1):
19         angle = compute_angle(coordinate_of_nodes[i-1],
coordinate_of_nodes[i], coordinate_of_nodes[i+1])
20         if angle != 180:
21             number_of_angle_changes += 1
22
23     return number_of_angle_changes

```

Listing 10: Code snippet the metric Change in angle

```

1  def calculate_direction(a: int, b: int) -> str:
2      """
3      Calculate the direction of the move from a to b.
4
5      Parameters
6      -----
7      a : int
8          Starting index.
9      b : int
10         Ending index.
11
12     Returns
13     -----
14     str
15         Direction of the move: "horizontal", "vertical", "diagonal", or
"same".

```

```

16     """
17     row_a, col_a = asp.row(a), asp.col(a)
18     row_b, col_b = asp.row(b), asp.col(b)
19     if row_a == row_b and col_a != col_b:
20         return "horizontal"
21     elif col_a == col_b and row_a != row_b:
22         return "vertical"
23     elif row_a != row_b and col_a != col_b:
24         return "diagonal"
25     else:
26         return "same"

```

Listing 11: Code Snippet for the metric change in direction

```

1  def complexity_score(pattern: tuple[int, ...]) -> int:
2      """
3      Compute the complexity score of a pattern using the metrics:
4          Length of pattern
5          Changes in direction
6          Changes in angle
7
8      Parameters
9      -----
10     pattern: tuple[int, ...]
11         A tuple of digits from 0 to 8 representing a swipe pattern.
12
13     Returns
14     -----
15     int
16         The complexity score of the pattern
17     """
18
19     score = 0
20
21     #Length of pattern
22     score += len(pattern)
23
24     # Changes in direction
25     for i in range(2, len(pattern)):
26         previous_direction = calculate_direction(pattern[i-2], pattern[i
27         -1])
28         current_direction = calculate_direction(pattern[i-1], pattern[i
29         ])
30
31         if previous_direction != current_direction:
32             score += 1
33
34     # Changes in angle
35     score += angle_change(pattern=pattern)

```

```
35     return score
```

Listing 12: Code snippet to compute pattern complexity score.

Code snippet to visualize a complex pattern

```
1  def random_pattern(length, pattern_dict):
2      '''
3      Parameters
4      -----
5      length : int
6          an integer length between 2 and 9
7      pattern_dict : dict
8          A dictionary mapping pattern lengths to sets of patterns
9          (like the output of generate_patterns, for example)
10
11      Returns
12      -----
13      A random choice of pattern from pattern_dict of the prescribed
14      length
15      '''
16      out = random.choice(list(pattern_dict[length]))
17      print(out)
18      return out
19
20
21  def draw_arrow(i, j):
22      '''
23      Parameters
24      -----
25      i : int
26          A node between 0 and 8.
27      j : int
28          A node between 0 and 8.
29
30      Returns
31      -----
32      None. Plots an arrow connecting node i to node j
33      '''
34      x1 = asp.col(i)
35      y1 = asp.row(i)
36      x2 = asp.col(j)
37      y2 = asp.row(j)
38      dx, dy = x2 - x1, y2 - y1
39
40      plt.arrow(x1, y1, dx, dy, head_width = 0.04, width = 0.01, ec = '
41      green')
```

```

42 def draw(path):
43     '''
44     Parameters
45     -----
46     path : tuple
47         A tuple of integers representing a swipe pattern
48
49     Returns
50     -----
51     None. Plots a visualization of the input pattern
52     '''
53
54     #Clear any existing plots
55     plt.clf()
56
57     #Draw the 9 dots representing the grid
58     for i in range(0,3):
59         for j in range(0,3):
60             plt.scatter(i, j, s=200, c='black', edgecolors='black')
61
62     #Invert the y-axis
63     plt.ylim(2.1, -0.1)
64
65     #Can't draw a path of length less than 2
66     if len(path) < 2:
67         return
68
69     #Connect each pair of adjacent nodes with an arrow:
70     for i in range(len(path)-1):
71         draw_arrow(path[i], path[i+1])
72
73     #Display the result
74     plt.show()

```

Listing 13: Code snippet to visualize a complex pattern

```

1  if __name__ == '__main__':
2      from random import choice
3      pattern_dict = generate_complex_patterns(threshold=5)
4      random_length = choice(range(4,10))
5      pattern = random_pattern(length=random_length, pattern_dict=
pattern_dict)
6      draw(path=pattern)

```

Listing 14: Plot of complex pattern