Python Journey

Introductory Problem Set

April 14, 2025

Disclaimer

In this handout we assume that students:

- Have already been introduced to Python functions, conditionals, for and while loops.
- Program through scripts instead of using the console.

The objective of these exercises is to reach a simple and intuitive solution, which not always corresponds to the most efficient or complete. Besides, some of these problems have built-in solutions in Python, but for learning purposes, we encourage students to ”reprogram” said problems.

Exercise 1 - Even Odd

a) Develop a function that receives an integer and calculates whether it is odd or even.

```python
def is_even(n: int) -> bool:
  return n % 2 == 0
```

b) Develop a test function for the Even Odd problem.

```python
def test_even_odd():
  assert is_even(0)
  assert not is_even(1)
  assert is_even(2)
  assert not is_even(3)
  assert is_even(100)
  assert not is_even(101)
  print("test_even_odd ok")

test_even_odd()
```

Note that an assert statement is expecting a boolean value, which is exactly what is_even returns, as such, we can shorten $is\_even(0) == True$ to $is_even(0)$, and $is\_even(1) == False$ to $notis\_even(1)$. Moreover, the value in the assert statement must be true, otherwise the program execution will stop abruptly to signal an error was encountered.

Exercise 2 - Grade Calculator

The following grading system is commonly used in the United States of America:

Letter Grade	Number Grade
A	90-100
B	80-89
C	70-79
D	60-69
F	0-59

a) Develop a function that receives an integer and calculates the corresponding letter grade. **Remember** to verify if the number received is within the desired scale.

```python
def grade_calculator(n: float) -> str:
  if n < 0 or 100 < n:
    return "Invalid grade"

  n = round(n)

  if n < 60:
    return "F"
  elif n < 70:
    return "D"
  elif n < 80:
    return "C"
  elif n < 90:
    return "B"
  else:
    return "A"
```

b) Develop a test function for the Grade Calculator problem.

```python
def test_grade_calculator():
  assert grade_calculator(-1) == "Invalid grade"
  assert grade_calculator(101) == "Invalid grade"
  assert grade_calculator(40) == "F"
  assert grade_calculator(59.2) == "F"
  assert grade_calculator(59.8) == "D"
  assert grade_calculator(71) == "C"
  assert grade_calculator(81) == "B"
  assert grade_calculator(91) == "A"
  assert grade_calculator(100) == "A"
  print("test_grade_calculator ok")

test_grade_calculator()
```

c) Develop an interactive program for the Grade Calculator problem. The user should also be able to terminate the program, for instance with the word ”end”. **Remember** to verify if the input is valid.

```python
def interactive_grade_calculator():
  is_on = True
  while is_on:
    val = input("Enter a number from 0 to 100: ")
    if val == "end":
      is_on = False
    else:
      try:
        val = float(val)
        grade = grade_calculator(val)
        print(str(val) + " corresponds to " + grade)
      except:
        print("Not a number!")

interactive_grade_calculator()
```

Exercise 3 - List Summation

Let $a = a_0, ..., a_{n-1}$, then a is sequence with n elements, starting from index 0 to $n-1$. The sum of the elements in sequence a, can be expressed as:

$$\sum_{i=0}^{n-1} a_i = a_0 + ... + a_{n-1}$$

Note that sequences can be stored as lists in python.

a) Develop a function that receives a list of floats and calculates the sum of its elements.

```python
def summation(l: list[float]) -> float:
  res = 0
  for i in l:
    res += i #equivalent to: res = res + i
  return res
```

Note that in '*for i in l*' (line 3), i will contain the values of the elements of the list. For instance, if $l = [2, 1, 0]$, then i would start as 2, then after updated res, would become 1, and, finally, 0.

b) Develop a test function for the List Summation problem.

```python
def test_summation():
  assert summation([]) == 0
  assert summation([0]) == 0
  assert summation([1]) == 1
  assert summation([0,1,2]) == 3
  assert summation([2,1,0]) == 3
  assert summation([3,3,3]) == 9
  assert summation([2, 6.2, 9.1, 30, 10]) == 57.3
  print("test_summation ok")

test_summation()
```

c) **Challenge:** Repeat the previous subproblems, but only sum the even numbers in the list.

```python
def even_summation(l: list[int]) -> int:
  res = 0
  for i in l:
    if i % 2 == 0:
      res += i
  return res

def test_even_summation():
  assert even_summation([]) == 0
  assert even_summation([0]) == 0
  assert even_summation([1]) == 0
  assert even_summation([0,1,2]) == 2
  assert even_summation([2,1,0]) == 2
  assert even_summation([3,3,3]) == 0
  assert even_summation([2, 6, 9, 30, 10]) == 48
  print("test_even_summation ok")

  test_even_summation()
```

Note that in this variation of the problem, we choose to change the data type of the parameter from a list of floats to a list of integers, because the even or odd definition does not apply to real numbers. While Python does not necessarily forbid the use of the modulo operator with floats, it may result in undefined or strange behaviour, which is best avoided.

Exercise 4 - Vowel Count

a) Develop a function that calculates the number of vowels in a string. **Warning**: The comparison between strings is **case-sensitive**.

```python
def count_vowels(s: str) -> int:
  res = 0
  for c in s:
    c = c.lower() #or c = c.upper(), and change "a" to "A", etc.
    if c == "a" or c == "e" or c == "i" or c == "o" or c == "u":
      res += 1
  return res
```

Note that choosing to use lower-case is just a convention, upper-case is also valid. While the if-expression is quite verbose due to the 5 comparisons in succession, due to its simplicity it is perfectly acceptable. In Python, unlike most other languages, there exists a shortcut: *if c in* [”*a*”,”*e*”,”*i*”,”*o*”,”*u*”], which under the hood makes *c* iterate over the values of the list, basically hiding a for loop.

b) Develop a test function for the Vowel Count problem.

```python
def test_count_vowels():
  assert count_vowels("") == 0
  assert count_vowels("my") == 0
  assert count_vowels("aeiou") == 5
  assert count_vowels("Hi, John!") == 2
  assert count_vowels("Almonds") == 2
  print("test_count_vowels ok")

test_count_vowels()
```

Exercise 5 - Multiplication Table

Multiplication tables are often used when teaching students multiplication for the first time, one example being:

Tabuada do 2
$2 * 1 = 2$
$2 * 2 = 4$
$2 * 3 = 6$
$2 * 4 = 8$
$2 * 5 = 10$
$2 * 6 = 12$
$2 * 7 = 14$
$2 * 8 = 16$
$2 * 9 = 18$
$2 * 10 = 20$

a) Develop an interactive program that prints the multiplication table for numbers selected by the user one at a time. The user should also be able to terminate the program, for instance with the word "end". **Remember** to verify if the input is valid.

```python
def mult_table(n: int) -> None:
  for i in range(1, 11):
    print(str(n) + " * " + str(i) + " = " + str(n*i))

def interactive_mult_table():
  is_on = True
  while is_on:
    val = input("Enter an integer: ")
    if val == "end":
      is_on = False
    else:
      try:
        val = int(val)
        mult_table(val)
      except:
        print("Not an integer!")

interactive_mult_table()
```

Exercise 6 - Fibonacci

There are innumerous ways to calculate the n^{th} number in the Fibonacci sequence. However, in this exercise students should use the iterative method, which means starting from the first and second Fibonacci numbers until one reaches the n^{th} number, using a loop.

Fibonacci sequence: $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...$

a) Develop a function that calculates the n^{th} Fibonacci number.

```
def fib_for(n: int) -> int:
  if n < 0:
    return
  elif n <= 1:
    return n

  a = 0
  b = 1

  for _ in range(1, n): # goes from 1 to n-1, meaning n-1 iterations
    sum = a+b
    a = b
    b = sum

  return b
```

Note that in this context we are not particularly interested in the value of the "loop" variable, which is omitted with _, or those specific starting and ending values of the range, instead we just need to perform a given number of iterations, this being $n-1$, because we start with Fib_1, and to calculate Fib_2 we need one iteration, so to calculate Fib_n, we need $n-1$ iterations. Finally, one particularly of this solution is that we return None (omitted) if the number is negative.

b) Develop a test function for the Fibonacci problem.

```
def test_fib_for():
  assert fib_for(-1) == None
  assert fib_for(0) == 0
  assert fib_for(1) == 1
  assert fib_for(2) == 1
  assert fib_for(3) == 2
  assert fib_for(4) == 3
  assert fib_for(5) == 5
  assert fib_for(10) == 55
  assert fib_for(18) == 2584
  print("test_fib_for ok")

test_fib_for()
```

c) **Challenge:** Repeat the previous subproblems for the Tribonacci problem, which instead of summing the two previous number, sums the three previous numbers. The Tribonacci sequence is as follows: $0, 1, 1, 2, 4, 7, 13, 24, 44, 81, ...$

```python
def tribonacci_for(n: int) -> int:
  if n < 0:
    return

  if n <= 1:
    return n

  a = 0
  b = 1
  c = 1

  for _ in range(2, n):
    sum = a+b+c
    a = b
    b = c
    c = sum

  return c

def test_tribonacci_for():
  assert tribonacci_for(-1) == None
  assert tribonacci_for(0) == 0
  assert tribonacci_for(1) == 1
  assert tribonacci_for(2) == 1
  assert tribonacci_for(3) == 2
  assert tribonacci_for(4) == 4
  assert tribonacci_for(5) == 7
  assert tribonacci_for(10) == 149
  assert tribonacci_for(18) == 19513
  print("test_tribonacci_for ok")

test_tribonacci_for()
```

Note that this implementation is quite similar to the Fibonacci problem, the biggest differences are the introduction of a third variable (c), and the range goes from 2 to $n-1$, which corresponds to $n-2$ iterations, because we start with $Trib_2$, and to calculate $Trib_3$ we need one iteration, so to calculate $Trib_n$, we need $n-2$ iterations.

Exercise 7 - Factorial

Similar to the Fibonacci sequence, there are various ways to calculate the factorial of a number, and once more, the objective will be to use its iterative version. However, in this case, one may iterate over the loop forward or backwards.

$$5! = 5 * 4 * 3 * 2 * 1 = 120$$

a) Develop a function that calculates the factorial of a number n by looping forward, i.e., from 1 to n.

```python
def factorial_forward(n: int) -> int:
  if n < 0:
    return -1

  res = 1
  for e in range(1, n+1):
    res = res * e
  return res
```

Note that 1 is the multiplicative neutral element, therefore, we initialize the result variable (*res*) with 1. The by multiplying every integer number from 1 to n (since range(1, n+1) goes from 1 to n).

b) Develop a function that calculates the factorial of a number n by looping backwards, i.e., from n to 1.

```python
def factorial_backwards(n: int) -> int:
  if n < 0:
    return -1

  res = 1
  for e in range(n, 0, -1):
    res = res * e
  return res
```

Note that 1 is the multiplicative neutral element, therefore, we initialize the result variable (*res*) with 1. By applying a negative step our range does backwards from n to 1, since the last element of the range is excluded (i.e. 0).

c) Develop a test function for the Factorial problem.

```python
def test_factorial():
  assert factorial_forward(0) == 1
  assert factorial_forward(1) == 1
  assert factorial_forward(2) == 2
  assert factorial_forward(3) == 6
  assert factorial_forward(4) == 24
  assert factorial_forward(5) == 120
  assert factorial_forward(10) == 3628800

  assert factorial_backwards(0) == 1
  assert factorial_backwards(1) == 1
  assert factorial_backwards(2) == 2
  assert factorial_backwards(3) == 6
  assert factorial_backwards(4) == 24
  assert factorial_backwards(5) == 120
  assert factorial_backwards(10) == 3628800
  print("test_factorial ok")

test_factorial()
```

Exercise 8 - Maximum Element

a) Develop a function that calculates the maximum element of a list of floats.
Suggestion: The result variable can be initialized as $float(-inf)$

```
def maximum(l: list[float]) -> float:
  res = float('-inf')
  for e in l:
    if e > res:
      res = e
  return res
```

Note that we want to initialize the result with the smallest amount possible, otherwise, the result might be altered unknowingly. For instance, if *res* starts at 0, then the maximum value of a list with just negative numbers would wrongly be zero. Alternatively, *res* could have been initialized with the first element of the list and the iteration start from the 2nd position onwards.

b) Develop a test function for the Maximum problem.

```
def test_maximum():
  assert maximum([]) == float('-inf')
  assert maximum([0]) == 0
  assert maximum([1]) == 1
  assert maximum([1, 2]) == 2
  assert maximum([2, 1]) == 2
  assert maximum([1, 7, 10, 4, 6, 12, 9]) == 12
  print("test_maximum ok")

test_maximum()
```

c) Repeat the previous subproblems for the minimum element.
Suggestion: The result variable can be initialized as $float(inf)$

```
def minimum(l: list[float]) -> float:
  res = float('inf')
  for e in l:
    if e < res:
      res = e
  return res

def test_minimum():
  assert minimum([]) == float('inf')
  assert minimum([0]) == 0
  assert minimum([1]) == 1
  assert minimum([1, 2]) == 1
  assert minimum([2, 1]) == 1
  assert minimum([1, 7, -10, 4, 6, 12, 9]) == -10
  print("test_minimum ok")

test_minimum()
```

Note that for the minimum element of a list, the result variable should be initialized with the highest value possible.

d) Challenge: Repeat the previous subproblems for the second-highest value of a list.

```
def second_maximum(l: list[float]) -> float:
  if len(l) < 2:
    return float('-inf')

  max = l[0]
  res = float('-inf')
  for e in l:
    if e > max:
      res = max
      max = e
    elif max > e and e > res:
      res = e
  return res

def test_second_maximum():
  assert second_maximum([]) == float('-inf')
  assert second_maximum([0]) == float('-inf')
  assert second_maximum([1, 2]) == 1
  assert second_maximum([2, 1]) == 1
  assert second_maximum([1, 7, 10, 4, 6, 12, 9]) == 10
  print("test_second_maximum ok")

test_second_maximum()
```

Note that calculating the second maximum element is slightly harder, and to do so we need variables for both the maximum and the second-highest values. This problem has an easy solution, which would be to loop through the list twice, once to find the maximum, and the second to find the second-highest value. On the other hand, our solution only iterates over the list once. There are two situations, where the second-highest candidate can be found, either we find a better candidate for the maximum value (if-statement), and the previous maximum candidate becomes the second-highest candidate, or, when we find a value that is superior to our second-highest candidate but not to the maximum value.

```
def second_minimum(l: list[float]) -> float:
  if len(l) < 2:
    return float('inf')

  min = l[0]
  res = float('inf')
  for e in l:
    if e < min:
      res = min
      min = e
    elif min < e and e < res:
      res = e
  return res
```

Note that calculating the second-lowest value follows a similar logic to calculating the second-highest, just beware of the inequality signs.

```python
def test_second_minimum():
  assert second_minimum([]) == float('inf')
  assert second_minimum([0]) == float('inf')
  assert second_minimum([1, 2]) == 2
  assert second_minimum([2, 1]) == 2
  assert second_minimum([1, 7, 10, 4, 6, 12, 9]) == 4
  print("test_second_minimum ok")

test_second_minimum()
```

Exercise 9 - Prime Numbers

A prime number is any integer greater than one and only be divisible by itself and one. The first five prime numbers are: $2, 3, 5, 7, 11$.

a) Develop a function that calculates if a given integer is a prime number.

```python
def is_prime(n: int) -> bool:
  if n <= 1:
    return False

  for i in range(2, n):
    if n % i == 0:
      return False

  return True
```

Note that for any positive integer, its divisors can be found between 1 and itself, and a number is divisible by another if their modulo is zero. So, if the modulo of the prime candidate by a number is zero in the range of 2 to $n-1$, then it is not prime, in which case we use *return* to terminate early. If the loop terminates without returning early, then it means no divisor in the range of 2 to $n-1$ exists, therefore it is prime. One deliberate "mistake" in the previous implementation is that we check in the range from 2 to $n-1$, however, theoretical results state that it suffices to check in the range from 2 to $\sqrt{n}$, however for simplicity and in the case of some students being unaware of that property, we decided to cover the full range.

b) Develop a test function for the Prime Numbers problem.

```python
def test_is_prime():
  assert not is_prime(1)
  assert is_prime(2)
  assert is_prime(3)
  assert not is_prime(4)
  assert is_prime(97)
  print("test_is_prime ok")

test_is_prime()
```

Exercise 10 - Common Elements

a) Develop a function that calculates the number of common elements at the **same position** of two lists. **Warning:** the two lists may be of different sizes, as such, beware of indexes out of bound.

```python
def count_eq_in_pos(l1, l2: list) -> int:
  size = min(len(l1), len(l2))

  res = 0
  for i in range(size):
    if l1[i] == l2[i]:
      res += 1

  return res
```

Note that accessing an invalid position of a list stops the program execution and displays the error "list index out of range", therefore we may only go until the end of the smallest array.

b) Develop a test function for the Common Elements problem.

```python
def test_count_eq_in_pos():
  assert count_eq_in_pos([], []) == 0
  assert count_eq_in_pos([2], [2]) == 1
  assert count_eq_in_pos([5], [9]) == 0
  assert count_eq_in_pos([2, 5], [2]) == 1
  assert count_eq_in_pos([1,2,3], [3,2,1]) == 1
  assert count_eq_in_pos([1,2,3], [1,2,3]) == 3
  print("test_count_eq_in_pos ok")

test_count_eq_in_pos()
```

Exercise 11 - Digit Sum

a) Develop a function that calculates the sum of all digits of an integer. **Suggestion:** Convert the integer to a string and only consider non-negative values.

```python
def sum_digits(n: int):
  digits = str(n)
  sum = 0
  for d in digits:
    sum = sum + int(d)
  return sum
```

Note that by considering the integer as a string, the problem becomes quite simple, since strings function similarly to lists in Python, in other words, the *for d in digits* expression separates the string into sub-strings containing just a single character (*d*), corresponding to a single digit. Now that we can consider each digit individually as a string, we just need to convert it to integer and perform the sum.

b) Develop a test function for the Digit Sum problem.

```python
def test_sum_digits():
  assert sum_digits(0) == 0
  assert sum_digits(1) == 1
  assert sum_digits(11) == 2
  assert sum_digits(123) == 6
  assert sum_digits(1234) == 10
  assert sum_digits(96) == 15
  assert sum_digits(100) == 1
  assert sum_digits(2718) == 18
  print("test_sum_digits ok")

test_sum_digits()
```

Exercise 12 - Digital Root

The digital root of a number is the repeated sum of digits until the result only contains a single digit. For instance:

$$\text{Digit sum of } 9045 = 18, \text{ because } 9+0+4+5 = 18$$

$$\text{Digital root of } 9045 = 9, \text{ because } 9+0+4+5 = 18 \text{ and } 1+8 = 9$$

a) Develop a function that calculates the digital root of an integer. **Suggestion:** Use the function from the previous exercise until the stopping condition is met.

```python
def digital_root(n: int):
  res = n
  while res >= 10:
    res = sum_digits(res)
  return res
```

Note that the digital root is the repeated sum of digits, ensuring that the final result has a single digit. As such, the simplest solution makes use of the previously developed function, inside a while loop checking for results with more than one digit. By attributing n to the res variable, one positive consequence is that we may skip the loop for values of n smaller than 10.

b) Develop a test function for the Digit Root problem.

```python
def test_digital_root():
  assert digital_root(0) == 0
  assert digital_root(1) == 1
  assert digital_root(11) == 2
  assert digital_root(123) == 6
  assert digital_root(1234) == 1
  assert digital_root(96) == 6
  assert digital_root(100) == 1
  assert digital_root(2718) == 9
  print("test_digital_root ok")

test_digital_root()
```

Exercise 13 - Count Character

a) Develop a function that calculates the number of times a character occurs in a string. **Warning:** Remember that python does not support characters directly like other languages, as such, one must use a string and guarantee that it has length of one.

```
def count_char(c: str, s: str) -> int:
  if len(c) != 1:
    return -1

  res = 0
  for e in s:
    if e == c:
      res += 1
  return res
```

Note that when variable c holds a string with length different than 1, i.e. it is not a character, then we return -1, which encodes to an invalid result. Since strings in Python function similarly to lists, then it is possible to separate a string into sub-strings of just one character, using *for e in s*.

b) Develop a test function for the Count Character problem.

```
def test_count_char():
  assert count_char("c", "count count count") == 3
  assert count_char("s", "count count count") == 0
  assert count_char("c", "") == 0
  assert count_char("as", "as usual") == -1
  assert count_char("", "count count count") == -1
  print("count_char ok")

test_count_char()
```

Exercise 14 - List Filter

a) Develop a function that filters the numbers from a list that are inferior to a certain threshold. **Suggestion:** The result should be a new list.

```
def list_filter(l: list[float], n: float) -> list[float]:
  res = []
  for e in l:
    if e >= n:
      res.append(e)
  return res
```

b) Develop a test function for the Inversion problem.

```
def test_list_filter():
  assert list_filter([], 0) == []
  assert list_filter([1,2,3,4], 3) == [3,4]
  assert list_filter([1,2,3,4], 5) == []
  assert list_filter([1,2,3,4], 0) == [1,2,3,4]
  print("test_list_filter ok")

test_list_filter()
```

Exercise 15 - Inversion

a) Develop a function that inverts the order of the elements in a list. **Suggestion:** The result should be a new list.

```
def invert_list(l: list) -> list:
  res = []
  size = len(l)
  for i in range(size):
    res.append(l[size-i-1])
  return res
```

Note that the expression $size - i - 1$, with $i = 0, ..., size - 1$, will result in accessing the positions of the list from end to start. Similarly, one could use *for* i in range($size - 1, -1, -1$), which is equivalent to $i = size - 1, ..., 0$ (Remember that the range function terminates one value earlier that what is passed there).

b) Develop a test function for the Inversion problem.

```
def test_invert_list():
  assert invert_list([]) == []
  assert invert_list([1]) == [1]
  assert invert_list([1, 1]) == [1, 1]
  assert invert_list([1, 2]) == [2,1]
  assert invert_list([1,2,3]) == [3,2,1]
  assert invert_list([1, 6, 8, 9, 2, 3, 5, 7]) == [7, 5, 3, 2, 9, 8, 6, 1]
  print("test_invert_list ok")

test_invert_list()
```

Exercise 16 - Palindromes

A palindrome is a sequence of symbols that can be read forward and backwards in the same way. However, one may usually find that some liberties are taken, for instance with punctuation or formatting. For simplicity, we recommend only considering truly palindromic sequences of characters, i.e. the characters must exactly match in both ways, with the only exception being the letter cases, if the student wishes.

By our definition:

- "Stop pots" is truly palindromic, if one ignores letter cases;
- "Never odd or even" is not truly palindromic, since the 5th character from the start is "r" and from the back is a space.

a) Develop a function that determines whether a string is truly palindromic.

```
def is_palindrome(s: str) -> bool:
  size = len(s)
  if size == 0:
    return True

  for i in range(size//2 + 1):
    if s[i] != s[size - i - 1]:
      return False
  return True
```

Note that since we are comparing the n^{th} first and last positions of the string, then it suffices to loop through the first half of the loop, doing the second half would just repeat the previous comparisons. The halfway point is calculated with the integer division (//) since the range does not allow floats as parameters. A particularity of the integer division is that we must compensate the remainder, for instance, consider a list of size 7, then $7//2 = 3$, which would mean that the halfway point is missing, since $range(3) = 0, 1, 2$, instead we would like $7//2 + 1 = 4$, since $range(4) = 0, 1, 2, 3$

b) Develop a test function for the Palindromes problem.

```
def test_is_palindrome():
  assert is_palindrome("")
  assert is_palindrome("aa")
  assert not is_palindrome("ab")
  assert is_palindrome("aba")
  assert not is_palindrome("abb")
  assert is_palindrome("stop pots")
  print("test_is_palindrome ok")

test_is_palindrome()
```

Exercise 17 - Remove Element

a) Develop a function that removes a given element from a list. **Warning:** The element may be repeated multiple times in the list. **Suggestion:** The result should be in a new list.

```
def remove_elem(l: list[int], n: int) -> list[int]:
  res = []
  for e in l:
    if e != n:
      res.append(e)
  return res
```

b) Develop a test function for the Common Elements problem.

```
def test_remove_elem():
  assert remove_elem([1,2,3], 1) == [2,3]
  assert remove_elem([], 1) == []
  assert remove_elem([1,2,3], 4) == [1,2,3]
  assert remove_elem([2], 2) == []
  assert remove_elem([1,0,1,0,1,0,1], 0) == [1,1,1,1]
  print("test_remove_elem ok")

test_remove_elem()
```

Exercise 18 - Remove Duplicates

a) Develop a function that removes the elements that are duplicated in a list. **Suggestion:** The result should be a new list.

```
def remove_dups(l: list) -> list:
  res = []
  for e in l:
    if not e in res:
      res.append(e)
  return res
```

Note that the keyword *in* used in the if-statement, works as a hidden loop, i.e., for element e, Python will check behind the hood if e is different from the elements of res one by one.

b) Develop a test function for the Remove Duplicates problem.

```
def test_remove_dups():
  assert remove_dups([]) == []
  assert remove_dups([1]) == [1]
  assert remove_dups([1, 1]) == [1]
  assert remove_dups([2, 2, 2]) == [2]
  assert remove_dups([1, 2, 3]) == [1, 2, 3]
  assert remove_dups([1, 3, 2, 3, 4, 1, 2, 5, 5, 2, 1, 3]) == [1, 3, 2, 4, 5]
  print("test_remove_dups ok")

test_remove_dups()
```

Exercise 19 - Linear Search

Linear Search is the most intuitive and simple searching algorithm. It consists in going from position to position in a list and checking whether the element stored there is the desired element. The algorithm returns a boolean to indicate whether it was found.

a) Develop a function that implements linear search.

```
def linear_search(l: list[float], n: float) -> list[float]:
  for e in l:
    if e == n:
      return True
  return False
```

b) Develop a test function for the Linear Search problem.

```
def test_linear_search():
  assert linear_search([1,2,3], 2)
  assert not linear_search([1,2,3], 4)
  assert not linear_search([0,1,2,3,4,5,6,7,8,9], 10)
  assert linear_search([0,1,2,3,4,5,6,7,8,9], 0)
  print("test_linear_search ok")

test_linear_search()
```

Exercise 20 - Sorted

a) Develop a function to check if a list is sorted **ascendingly**.

```
def is_sorted(l: list[int]) -> bool:
  for i in range(len(l) - 1):
    if l[i] > l[i+1]:
      return False
  return True
```

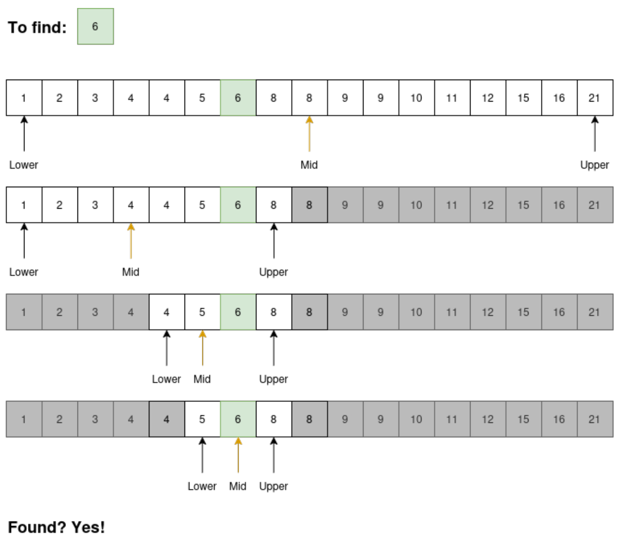
b) Develop a test function for the Sorted problem.

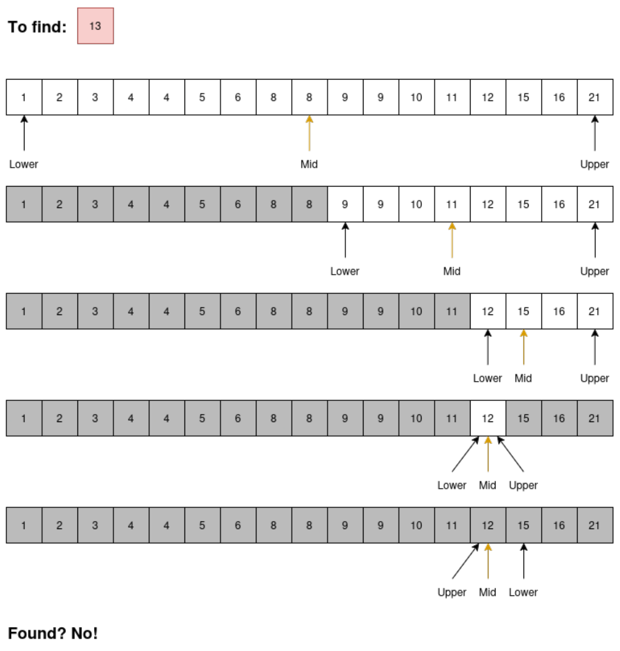
```
def test_is_sorted():
  assert is_sorted([])
  assert is_sorted([1])
  assert not is_sorted([2,1])
  assert is_sorted([1, 3, 5])
  assert not is_sorted([2, 6, 4, 8])
  assert not is_sorted([2, 2, 3, 2])
  assert is_sorted([1, 1, 1, 1])
  assert is_sorted([-1, -1, 0, 0, 1, 1])
  print("test_is_sorted ok")

test_is_sorted()
```

Exercise 21 - Binary Search

Imagine that we have a sorted list, and we are looking for a specific value, would linear search be efficient? Possibly not, we might be looking for large value present at the end of a very large list. A common and efficient solution used in various fields of Engineering is to divide the domain in half, and if possible exclude one of the two halves. Binary Search does exactly that, since the list must be sorted beforehand, by checking the value at the middle point, we can restrict the domain in half in case the middle point is not the value we are looking for. This process can be repeated while the domain is valid. For clarity, see the following Images:





a) Complete the following function:

```
def bin_search(l:list[int], n: int) -> bool:
  assert is_sorted(l)

  lower = 0
  upper = len(l) - 1

  while lower <= upper:
    mid = (lower + upper)//2
    if l[mid] == n:
      return True
    elif l[mid] > n:
      upper = mid - 1
    else:
      lower = mid + 1

  return False
```

Note that when n is equal to the midway point, we return $True$, because we found the correct value. On the other hand, if the while loop is broken (i.e. *lower* becomes greater than *upper*, meaning a nonsensical interval), then it means the value was not present in the list, therefore we wish to return $False$. Regarding the initial boundaries, as excepted $lower = 0$ and $upper = len(l) - 1$, this corresponds to the full interval of indexes for list l. The *mid* variable can be calculated as half of the sum of the *lower* and *upper* boundaries (middle point formula). Since, *mid* is used for accessing various of a list, then it must be an integer, thus we use the integer division. In intervals with and odd number of elements, we need to choose between the two middle elements, by disregarding the remainder (what we do here), we select the first middle element, while considering the remainder (with value 1, since we divide by 2), then we would be selecting the second middle element.

b) Develop a test function for the Binary Search problem.

```
def test_bin_search():
  assert bin_search([1,2,3], 2)
  assert not bin_search([1,2,3], 4)
  assert not bin_search([0,1,2,3,4,5,6,7,8,9], 10)
  assert bin_search([0,1,2,3,4,5,6,7,8,9], 0)
  print("test_bin_search ok")

test_bin_search()
```

Exercise 22 - Matrix Sum

Let A and B, be two $n * n$ matrices, then:

$$A + B = \begin{bmatrix} a_{11} & \dots & a_{1n} \\ \dots & \dots & \dots \\ a_{n1} & \dots & a_{nn} \end{bmatrix} + \begin{bmatrix} b_{11} & \dots & b_{1n} \\ \dots & \dots & \dots \\ b_{n1} & \dots & b_{nn} \end{bmatrix} = \begin{bmatrix} a_{11} + b_{11} & \dots & a_{1n} + b_{1n} \\ \dots & \dots & \dots \\ a_{n1} + b_{n1} & \dots & a_{nn} + b_{nn} \end{bmatrix}$$

For instance:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} + \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 2 & 3 \\ 4 & 6 & 6 \\ 7 & 8 & 10 \end{bmatrix}$$

a) Develop a function to check if the matrix has the same number of lines and columns.

```python
def is_n_n(mtx: list[list[float]]) -> bool:
  line_num = len(mtx)
  for i in range(line_num):
    if len(mtx[i]) != line_num:
      return False
  return True
```

b) Develop a test function for the previous subproblem

```python
def test_is_n_n():
  assert is_n_n([])
  assert is_n_n([[1]])
  assert is_n_n([[1,2], [3,4]])
  assert not is_n_n([[1,2]])
  assert not is_n_n([[1,2], [3]])
  print("test_is_n_n ok")

test_is_n_n()
```

c) Develop a function that calculates the sum of two matrices. **Suggestions:** Guarantee that both matrices are $n * n$ and the result should be a new matrix.

```python
def matrix_sum(mtx1, mtx2: list[list[float]]) -> list[list[float]]:
  assert is_n_n(mtx1)
  assert is_n_n(mtx2)
  assert len(mtx1) == len(mtx2)

  res = []
  n = len(mtx1)
  for i in range(n):
    line = []
    for j in range(n):
      line.append(mtx1[i][j] + mtx2[i][j])
    res.append(line)

  return res
```

d) Develop a test function for the Matrix Sum problem.

```
def test_matrix_sum():
  assert matrix_sum([[15]], [[15]]) == [[30]]
  assert matrix_sum([[1,1], [1,1]], [[2,2], [2,2]]) == [[3,3], [3,3]]
  assert matrix_sum([[1,2], [3,4]], [[5,6], [7,8]]) == [[6,8], [10,12]]
  assert matrix_sum([[1,1], [1,1]], [[0,0], [0,0]]) != [[3,3], [3,3]]
  print("test_matrix_sum ok")

test_matrix_sum()
```

Exercise 23 - Matrix Transposition

Let A be $n * m$ matrix, and if:

$$A = \begin{bmatrix} a_{11} & \dots & a_{1m} \\ \dots & \dots & \dots \\ a_{n1} & \dots & a_{nm} \end{bmatrix}, \text{ then } A^T = \begin{bmatrix} a_{11} & \dots & a_{n1} \\ \dots & \dots & \dots \\ a_{1m} & \dots & a_{nn} \end{bmatrix}$$

For instance, if:

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}, \text{ then } A^T = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix},$$

a) Develop a function to check if the list of lists has the same number of elements in every sub-list. In other words, if it is indeed an $n * m$ matrix, and every column has exactly m elements.

```
def is_matrix(mtx: list[list[float]]) -> bool:
  line_num = len(mtx)
  if line_num == 0:
    return True

  col_num = len(mtx[0])
  for i in range(1, line_num):
    if col_num != len(mtx[i]):
      return False
  return True
```

b) Develop a test function for the previous subproblem

```
def test_is_matrix():
  assert is_matrix([])
  assert is_matrix([[1]])
  assert is_matrix([[1,2,3,4]])
  assert not is_matrix([[1], [1,2]])
  assert is_matrix([[1,2], [3,4]])
  assert is_matrix([[1,3,5], [2,4,6]])
  print("test_is_matrix ok")

test_is_matrix()
```

c) Develop a function that calculates the transpose of a matrix. **Suggestions:** Guarantee that it is a valid matrix and the result should be a new matrix.

```
def transpose(mtx: list[list[float]]) -> list[list[float]]:
  assert is_matrix(mtx)

  num_line = len(mtx)
  num_col = len(mtx[0])

  res = []
  for j in range(num_col):
    line = []
    for i in range(num_line):
      line.append(mtx[i][j])
    res.append(line)

  return res
```

d) Develop a test function for the Matrix Transposition problem.

```
def test_transpose():
  assert transpose([[1,2], [2,1]]) == [[1,2], [2,1]]
  assert transpose([[1,2], [3,4]]) == [[1,3], [2,4]]
  assert transpose([[1,2], [3,4], [5,6]]) == [[1, 3, 5], [2, 4, 6]]
  print("test_transpose ok")

test_transpose()
```

Exercise 24 - Interactive Average

a) Complete the following program, which calculates and prints the average of all numbers introduced by the user beforehand:

```
def interactive_avg():
  is_on = True
  sum = 0
  count = 0
  while is_on:
    val = input("Write a number: ")
    if val == "end":
      is_on = False
    else:
      try:
        sum += float(val)
        count += 1
        print("Average: ", str(sum/count))
      except ValueError:
        print("Not a number!")

interactive_avg()
```

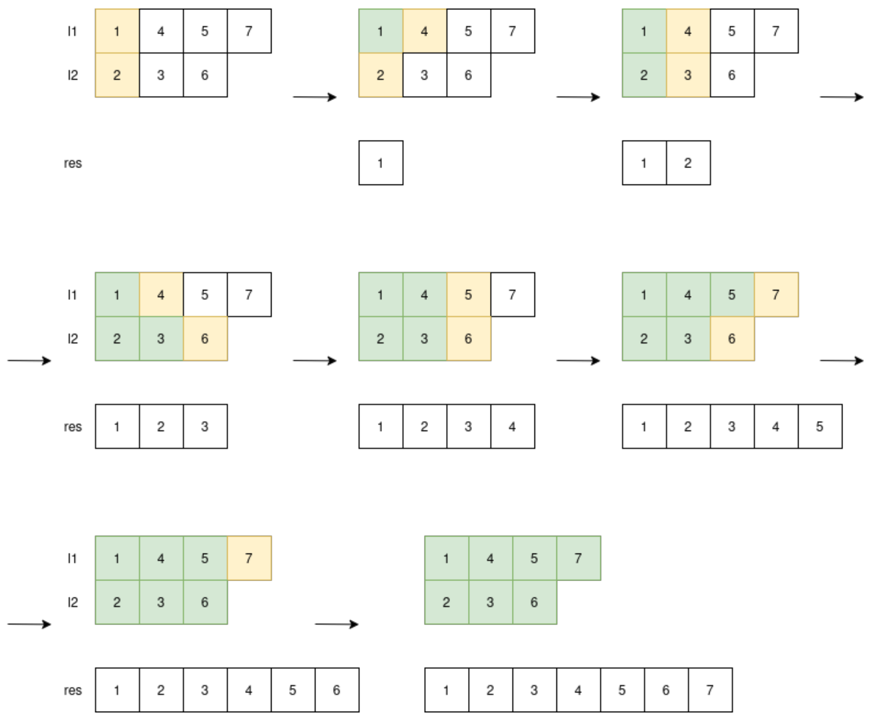
Exercise 25 - Sorted Concatenation

Concatenation commonly refers to the operation of joining two lists, usually the second list is placed at the end of the first. In Python this can be achieved with the 'extend' operation.

```
l1 = [1, 4, 5, 7]
l2 = [2, 3, 6]
l1.extend(l2)
# l1 = [1, 4, 5, 7, 2, 3, 6]

l3 = [1, 2, 3, 4]
l4 = [5, 6, 7]
l3.extend(l4)
# l4 = [1, 2, 3, 4, 5, 6, 7]
```

However, by naively concatenating two sorted lists the resulting list may lose that property (unless every element of $l1$ is lesser or equal that every element of $l2$). One possible solution would be to naively concatenate the two list then sort the result, which is not an interesting exercise, since Python offers a built-in function for sorting lists. Instead, we propose to build a new list (unlike 'extend' which alters the first list) and insert the element in order, for instance, for $l1$ and $l2$:



Suggestion: To achieve this, use a while loop that iterates over both lists at the same time, but only advance one element of one of the lists at a time.

a) Develop a function that performs the sorted concatenation of two lists.

```
def sorted_concat(l1, l2: list[float]) -> list[float]:
  assert is_sorted(l1) and is_sorted(l2)
  size1 = len(l1)
  size2 = len(l2)
  i1 = 0
  i2 = 0
  res = []
  while i1 < size1 or i2 < size2:
    if i1 >= size1 or (i2 < size2 and l1[i1] >= l2[i2]):
      res.append(l2[i2])
      i2 += 1
    elif i2 >= size2 or (i1 < size1 and l1[i1] < l2[i2]):
      res.append(l1[i1])
      i1 += 1
  return res
```

b) Develop a test function for the Sorted Concatenation problem.

```
def test_sorted_concat():
  assert sorted_concat([1], [2]) == [1,2]
  assert sorted_concat([1,3], [2]) == [1,2,3]
  assert sorted_concat([1,2,3], [2,4]) == [1,2,2,3,4]
  assert sorted_concat([0,2,4,5,6,7,10,12,13], [1,3,8,9,11]) == [0,1,2,3,4,5,6,7,8,9,10,11,12,13]
  print("test_sorted_concat ok")

test_sorted_concat()
```

Exercise 26 - Bank Account

Consider an interactive program that simulates a simple ATM to access a bank account, and contains the following operations:

Button	Operation
1	Check balance
2	Deposit
3	Withdraw
4	Help
5	Quit

No need to consider any kind of security or account management aspects, or other operations (Imagine that you have your own ATM at home, which only knows (one of) your account(s))

a) Develop a function for the help operation. It should print the necessary information about the operations available.

```
def bank_help():
  print("Available operations:")
  print("1. Get Balance")
  print("2. Deposit")
  print("3. Withdraw")
  print("4. Help")
  print("5. Quit")
```

b) Complete the following Deposit function:

```
def deposit(currBalance):
  amount = None
  while amount == None:
    try:
      amount = float(input("Deposit amount: "))
      if amount <= 0:
        print("Not a positive amount! Operation Cancelled")
        return currBalance
      else:
        return currBalance + amount
    except:
      print("Not a number!")
```

c) Complete the following Withdraw function:

```
def withdraw(currBalance):
  amount = None
  while amount == None:
    try:
      amount = float(input("Withdraw amount: "))
      if amount <= 0:
        print("Not a positive amount! Operation Cancelled")
        return currBalance
      elif amount > currBalance:
        print("Insufficient amount! Operation Cancelled")
        return currBalance
      else:
        return currBalance - amount
    except:
      print("Not a number!")
```

d) Develop the main function of the ATM. **Suggestions:**

- It should follow the interactive program pattern (indefinite while loop that can be terminated by the user's input)
- Despite the previous buttons being represented as numbers, we are not interested in their numerical value, as such, it suffices using them as a string (In other words, no need to use the try/except shown before to avoid trying to convert things that may not be numbers into integers or floats)
- In console-based programming, where we have no other visual Elements other than text, it is good practice to display the help information at the start of the interactive program without the user requesting
- Do not forget to print a message if the user inputs something that does not appear in the table.
- Use the previous functions.

```python
def bank_account():
  is_on = True
  balance = 0
  bank_help()
  while is_on:
    op = input("Choose an operation: ")
    if op == "1":
      print("Balance = " + str(balance))
    elif op == "2":
      balance = deposit(balance)
    elif op == "3":
      balance = withdraw(balance)
    elif op == "4":
      bank_help()
    elif op == "5":
      is_on = False
    else:
      print("Unknown operation!")

bank_account()
```