Homework 1 (100 points) **Due date:** September 19, 2025, 11:59 PM

Before you start…

Keep in mind these tips from your instructor:

- Outline a strategy before you start typing code: Going from instructions to code is a common practice among students, but this could lead to hours of frustration and anxiety that often result in academic dishonesty. Problem-solving skills are crucial for coding, if you are unable to implement a function, assume the computer understands English and prepare a course of action based on that fact (like pseudocode code or bullet points) so you can discuss it with the course staff or your peers. That is how you learn
- Write clean code: "It works" is not enough when it comes to coding in this course. In practice, programmers work in teams, so others will be reading your code often. Always do your best to write your code in the most concise and readable way that you can. Start the habit of clean coding now, as it will save you hours trying to decipher your work after you haven't looked at it in months!
- All functions that output a value must **return** it, not **print** it. Code will not receive credit if you use print to display the output. No function calls or testing code should be left in your final submission, your file should only contain your function definitions and implementations.

```
def foo(arg):
        final = value + arg
        return final
```

- **The use of *break* or *continue* statements, or recursive solutions is NOT allowed anywhere in this assignment.**
- Ask questions using our Homework 1 channel in Microsoft Teams

REMINDER: As you work on your coding assignments, it is important to remember that passing the examples provided does not guarantee full credit. While these examples can serve as a helpful starting point, it is ultimately your responsibility to thoroughly test your code and ensure that it is functioning correctly and meets all the requirements and specifications outlined in the assignment instructions. Failure to thoroughly test your code can result in incomplete or incorrect outputs, which will lead to deduction in points for each failed case.

Create additional test cases and share them with you classmates on Teams, we are in this together!

60% > Passing all test cases 40% > Clarity and design of your code

Question #1: `rectangle(perimeter, area)` **10 pts**

Returns the longest side of the rectangle with given perimeter and area if and only if both sides are integer lengths. In a rectangle $perimeter = 2w + 2h$ and $area = w * h$. Only one loop is allowed.

```
Preconditions and Postconditions
perimeter: int
area: int
Returns: int -> The longest side length for rectangle of integer lengths
         False -> No rectangle exists that meets the requirements
```

Allowed methods, operators, and libraries (if it is not in this list, you cannot use it):

- Loops, conditionals, all arithmetic operations
- math.sqrt from the math module
- Type conversion int()
- Integer division (also known as floor division), using the // operator. Floor division, as opposed to true division, discards any fractional result from the output.
  - 5//2 returns 2
- The built-in float.is_integer() method returns True if the float instance is finite with integral value, and False otherwise
  - 5.35.is_integer() returns False
  - 5.0.is_integer() returns True
- The built-in round(number, digits) method returns a floating-point number rounded to the specified number of decimals
  - round(3.14159, 1) returns 3.1
  - round(3.14159, 2) returns 3.14
  - round(3.14159, 0) returns 3.0

Examples:

```
>>> rectangle(14, 10)    # Represents a 2x5 rectangle
5                        # 5 is the larger of the two sides
>>> rectangle(25, 25)    # Represents a 2.5x10 rectangle
False                    # 2.5 is not an integer
```

Give me those digits….

When you are given a sequence of numbers such as [6, 7, 5], printing each element from right to left can be achieved with a simple loop:

```
lst = [6, 7, 5]
for i in range(len(lst)-1, -1, -1):
     print(lst[i])

5
7
6
```

But what if we want to print each digit from a given number such as 675? To start, integers do not have a `len` method to get the number of digits.

```
>>> len(675)
TypeError: object of type 'int' has no len()
```

Additionally, integers are not iterable!

```
for i in 675:
     print(i)
TypeError: 'int' object is not iterable
```

To process digits in a **positive integer**, modulo (%) 10 and floor division (//) by 10 can help us extract and remove the least significant digit:

```
>>> num = 675
>>> print(num%10)
5
>>> num = num//10
>>> num
67
>>> print(num%10)
7
>>> num = num//10
>>> num
6
>>> print(num%10)
6
>>> num = num//10
>>> num
0
```

Floor division by 10 removes the rightmost digit (675//10 = 67), while modulo 10 returns the rightmost digit (675%10 = 5). This set of operations combined with a `while` loop will allow you to traverse the integer from right to left.

Can you spot the continuation condition of the loop?

Question #2: `to_decimal(oct_num)` **10 pts**

Given a number *oct_num* in octal system (base 8), return *oct_num* in decimal system (base 10). In the octal number system, each digit represents a power of eight and it uses the digits 1 to 7. To convert a number represented in octal system to a number represented in decimal system, each digit must be multiplied by the appropriate power of eight. For example, given the octal number 206_8 results in the decimal number 134_{10}:

$$\begin{array}{ccc} 2 & 0 & 6 \\ 8^2 & 8^1 & 8^0 \end{array} \longrightarrow 206 = (2 \times 8^2) + (0 \times 8^1) + (6 \times 8^0) = 134$$

- You are NOT allowed to type convert *num* to a string <str(*num*)>, add the digits into a list to traverse or process the number, or to use of the math module

Preconditions and Postconditions

```
oct_num: int -> Positive number that always starts with a digit in range 1-7
Returns: int -> decimal representation of oct_num
```

Example:

```
>>> to_decimal(237)   # (8² * 2) + (8¹ * 3) + (8⁰ * 7)
159
>>> to_decimal(35)    # (8¹ * 3) + (8⁰ * 5)
29
```

Question #3: `has_hoagie(num)` **10 pts**

A given integer *num* has a hoagie if at least a digit in *num* is surrounded by two identical digits. This function takes an integer and returns True is *num* has a hoagie, False otherwise. Note that num//10 and num%10 will not behave as intended when *num* is negative, so make sure your process negative numbers as positive.

Tip: A number that has less than three digits cannot have a hoagie.

- You are NOT allowed to type convert *num* to a string <str(*num*)>, add the digits into a list to traverse or process the number, or to use of the math module

Preconditions and Postconditions

```
num: int -> any integer value
Returns: bool -> True is num has a hoagie, False otherwise
```

Example:

```
>>> has_hoagie(737)   # 737
True
>>> has_hoagie(35)
False
```

```
>>> has_hoagie(-6060)  # 606 or 060
True
>>> has_hoagie(-111)  # 111
True
>>> has_hoagie(6945)
False
```

Question #4: `is_identical(num_1, num_2)` **10 pts**

Given two positive numbers, num_1 and num_2, the function returns True if after removing all sequences of repeated digits from both integers and replacing it with only one instance, num_1 is equal to nu_2, False otherwise.

- You are NOT allowed to type convert *num* to a string <str(*num*)>, add the digits into a list to traverse or process the number, or to use of the math module

Preconditions and Postconditions

```
num_1: int -> positive integer
num_2: int -> positive integer
Returns: bool -> True is num_1==num_2 after removing sequences of repeated
digits, False otherwise
```

Example:

```
>>> is_identical(51111315, 51315) #1111 is replaced with 1, 51315 == 51315
True
>>> is_identical(7006600, 7706000) # 7060==7060
True
>>> is_identical(135, 765)
False
>>> is_identical(2023, 20)
False
```

Question #5: `hailstone(num)` **10 pts**

Returns the hailstone sequence starting at the given number until termination when *num* is 1. The hailstone sequence is generated as follows:

- If num is even, then the next number in the sequence is $num/2$.
- If num is odd, then the next number in the sequence is $3 * num + 1$.
- Continue this process until $num = 1$

The use of type conversion such as int(), and list comprehension syntax are not allowed.
Implement this function without redundancy (do not repeat the same operations in conditionals)

Preconditions and Postconditions

```
num: int -> An positive integer
Returns: list -> Hailstone sequence starting at num and ending at 1.
```

Examples:

```
>>> hailstone(5)
[5, 16, 8, 4, 2, 1]
>>> hailstone(6)
[6, 3, 10, 5, 16, 8, 4, 2, 1]
```

Question #6: `overloaded_add(d, key, value)` **10 pts**

Given a dictionary *d*, a *key*, and a *value*, this function adds the key:value pair to the dictionary *d*. If the key doesn't already exist in the dictionary, this function should simply add it. However, if the key already exists, then rather than overriding the previous value, the dictionary should start keeping a list of values for that key, in the order they are added.

Preconditions and Postconditions

```
d: dict -> You cannot make any assumptions about the size or contents of d
key: any immutable type
value: any
Returns: None
```

Examples:

```
>>> my_dict = {"Alice": "Engineer"}
>>> overloaded_add(my_dict, "Bob", "Manager")
>>> my_dict
{"Alice": "Engineer", "Bob": "Manager"}
>>> overloaded_add(my_dict, "Alice", "Sales")
>>> my_dict
{"Alice": ["Engineer", "Sales"], "Bob": "Manager"}
```

Allowed methods, operators, and libraries:

- type() or isinstance() methods
  `type([5.6]) == list  returns True`
  `isinstance([5.6], list)  returns True`
- list concatenation (+) or append()

Question #7: `by_department(d)` **10 pts**

Given a dictionary *d*, that contains employee information in the format `{employee_id: {'name': name, 'position': position, 'department': department}}`, returns a new dictionary with the employee information by department using the format `{department: [{'emp_id': employee_id, 'name': name, 'position': position}, {'emp_id': employee_id, 'name': name, 'position': position}]}`. Note that employee id

becomes the key and it disappears as a value in the inner dictionary. Do not mutate (change) the original dictionary.

Preconditions and Postconditions

d: dict with format {employee_id: {'name': name, 'position': position, 'department': department}, …}
Returns: dict with format {department: [{'emp_id': employee_id, 'name': name, 'position': position}, {'emp_id': employee_id, 'name': name, 'position': position}], …}

Examples:

```
>>> employees = {
        1: {'name': 'John Doe', 'position': 'Manager', 'department': 'Sales'},
        2: {'name': 'Sara Miller', 'position': 'Budget Advisor', 'department': 'Finance'},
        3: {'name': 'Jane Smith', 'position': 'Engineer', 'department': 'Engineering'},
        4: {'name': 'Bob Johnson', 'position': 'Analyst', 'department': 'Finance'},
        5: {'name': 'Clark Wayne', 'position': 'Senior Developer', 'department': 'Engineering'}
        }

>>> by_department(employees)   # spaced for readability
{'Sales': [
             {'emp_id': 1, 'name': 'John Doe', 'position': 'Manager'}
           ],
'Finance': [
                {'emp_id': 2, 'name': 'Sara Miller', 'position': 'Budget Advisor'},
                {'emp_id': 4, 'name': 'Bob Johnson', 'position': 'Analyst'}
            ],
'Engineering': [
                 {'emp_id': 3, 'name': 'Jane Smith', 'position': 'Engineer'},
                 {'emp_id': 5, 'name': 'Clark Wayne', 'position': 'Senior Developer'}
               ]
}
```

Reading files

You will be required to read data from .txt files. The starter code already contains the code to open and close the *txt* file (Files will be closed automatically at the end of `with` statement block or if any exceptions or errors occur in the process), just make sure the file is in the same directory as your .py file.
The given code to read the *txt* file uses read(), but there are three different ways you can read a file. You might choose any of the other two if it fits the outline of your program better. If you are new to reading files in Python, I suggest you try them all first using the read_files.py file so you can decide which one to use.

```
Example using words.txt:
ball
bats
bird
toys
treat
bin
tree
boy
```

Reading a text file using iteration

- The file is an iterable that generates one line at a time
- Each line generated is a string, ending with a newline (\n)

```
lines = []
with open(file_path) as file:
    for line in file:
         lines.append(line)
print(lines)
['ball\n', 'bats\n', 'bird\n', 'toys\n', 'treat\n', 'bin\n', 'tree\n', 'boy']
```

Reading a text file using read()

- Reads everything in the original file into one big string all the way to the end of the file.

```
with open(file_path) as file:
   contents = f.read()
print(contents)
'ball\nbats\nbird\ntoys\ntreat\nbin\ntree\nboy'
```

Reading a text file using readlines()

- Reads everything in the original file into a list of lines (similar to iteration that appends to a list)

```
with open(file_path) as file:
  contents = file.readlines()
print(contents)
['ball\n', 'bats\n', 'bird\n', 'toys\n', 'treat\n', 'bin\n', 'tree\n', 'boy']
```

Question #8 `successors(file_name)` **15 pts**

Returns a dictionary whose keys are included words (as is), and values are lists of unique successors to those words. The first word of the *txt* file is always a successor to ".". Be careful with punctuation. You can assume the text in the *txt* file does not contain any contraction words (isn't, don't, I'm, etc) but you cannot make assumptions about the spacing or the presence of non-alphanumerical characters. The starter code already contains the code to read the *txt* file, just make sure the file is in the same directory as your .py file. You must write your code after the file has been read to process the contents.

```
# Open the file and read the contents, the with statement ensures
  the file properly closed after the file operation finishes
with open(file_path) as file:
    contents = file.read() # reads the entire file, saving data in contents as string
```

`items.txt:`

We came to learn,eat some pizza and to have fun.
Maybe to learn how to make pizza too!

```
>>> print(contents)
'We came to learn,eat some pizza and to have fun.\nMaybe to learn how to make
pizza too!'
```

Hints:

- The str.isalnum() method returns True if all characters in a string are alphanumeric.
- When *x* = 'We', the type conversion list(*x*) produces the list ['W', 'e']

Preconditions and Postconditions

```
file_name: str -> A string that ends with .txt
Returns: dict -> Keys are all included words and punctuation marks, values all
successors of that key
```

Example:

```
Contents of items.txt:
We came to learn,eat some pizza and to have fun.
Maybe to learn how to make pizza too!

>>> out=successors('items.txt')
>>> out.keys()
dict_keys(['.', 'We', 'came', 'to', 'learn', ',', 'eat', 'some', 'pizza', 'and',
'have', 'fun', 'Maybe', 'how', 'make', 'too'])
>>> out['.']
['We', 'Maybe']
>>> out['to']
['learn', 'have', 'make']
>>> out['fun']
['.']
```

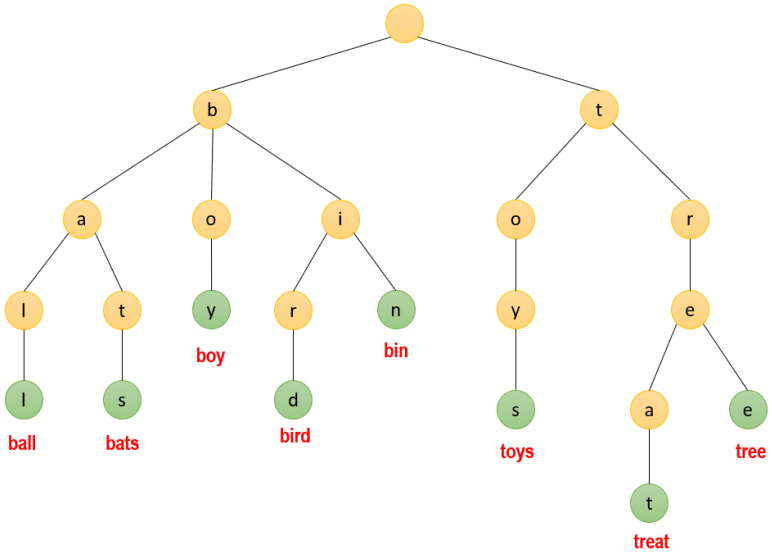
```
>>> out[',']
['eat']
>>> out        # Don't forget dictionaries are unsorted collections
{'.': ['We', 'Maybe'], 'We': ['came'], 'came': ['to'], 'to': ['learn', 'have',
'make'], 'learn': [',', 'how'], ',': ['eat'], 'eat': ['some'], 'some': ['pizza'],
'pizza': ['and', 'too'], 'and': ['to'], 'have': ['fun'], 'fun': ['.'], 'Maybe':
['to'], 'how': ['to'], 'make': ['pizza'], 'too': ['!']}
```

Allowed methods, operators, and libraries:

- str.split(sep) returns a list of the words in the string, using *sep* as the delimiter string
  - 'a b c'.split(' ') returns ['a', 'b', 'c']
  - 'a,b,c'.split(',') returns ['a', 'b', 'c']
  - 'a=b'.split('=') returns ['a', 'b']
- str.strip(), returns a copy of the string with the leading and trailing characters removed.
  - ' a , b '.strip() returns 'a , b'
  - ' a , b\n'.strip('\n') returns ' a , b'
- The str.join(iterable) method returns a string which is the concatenation of the strings in iterable
  - ''.join(['a', 'b', 'c']) returns 'abc'
  - '-'.join(['a', 'b', 'c']) returns 'a-b-c'
  - ' '.join(['a', 'b', 'c']) returns 'a b c'
- Loops and conditionals
- Append or list concatenation (+)

Question #9: Trie **15 pts**

In this question you will use dictionaries to implement a simple version of a data structure often used for storing large dictionaries of words efficiently, namely a trie or a prefix tree. The way a trie works, is that each level stores one letter of a word, and then links to another level. This combines words with like prefixes, which makes searching and storing the information much easier:



For the purposes of this assignment, we will consider each level of the structure a dictionary with keys that are the next letter in the word, and values that are dictionaries of the next level. In order to know when we have reached a word, we will add a key, **'word'**, with boolean value **True**, at any layer where the path there represents a full word.

For example, the word 'at' is represented `{'a': {'t': {'word': True}}}` and adding the word 'ate' gets us `{'a': {'t': {'word': True, 'e': {'word': True}}}}`

For a larger example, using the original trie example, we'd represent it with dictionaries as follows:

```
{'b': {'a': {
             'l': {'l': {'word': True}},
             't': {'s': {'word': True}}
            },
       'o': {'y': {'word': True}},
       'i': {'r': {'d': {'word': True}},
             'n': {'word': True}
            }
      },
't':  {
       'o': {'y': {'s': {'word': True}}},
       'r': {'e': {'a': {'t': {'word': True}},
                   'e': {'word': True}
                  }
            }
      }
}
```

The starter code already contains the code to read the *txt* file, just make sure the file is in the same directory as your .py file. You must write your code after the file has been read to process the contents.

```
Example of value for variable contents using list_of_words.txt:
ball
bats
bird
toys
treat
bin
tree
boy

>>> print(contents)
'ball\nbats\nbird\ntoys\ntreat\nbin\ntree\nboy'
```

addToTrie(trie, word) **7/15 pts**

Given a dictionary that represents a trie, adds word to the dictionary. This function mutates the given dictionary, in other words, it modifies trie.

Preconditions and Postconditions

```
trie: dict -> trie of words
word: str -> non-empty string
Returns: None
```

Examples:

```
>>> trie_dict = {'a' : {'word' : True,
                        'p' : {'p' : {'l' : {'e' : {'word' : True}}}},
                 'i' : {'word' : True}}}
>>> addToTrie(trie_dict, 'art')
>>> trie_dict
{'a': {'word': True, 'p': {'p': {'l': {'e': {'word': True}}}}, 'i': {'word':
True}, 'r': {'t': {'word': True}}}}
>>> addToTrie(trie_dict, 'moon')
>>> trie_dict
{'a': {'word': True, 'p': {'p': {'l': {'e': {'word': True}}}}, 'i': {'word':
True}, 'r': {'t': {'word': True}}}, 'm': {'o': {'o': {'n': {'word': True}}}}}
```

Allowed methods, operators, and libraries:

- slicing operator [:]

`createDictionaryTrie(file_name)` **4/15 pts**

Given the name of a text file with one word on each line, this function should construct and return a Trie representation of all of the words. You can assume there will be no duplicate words, however, words can be in both uppercase and lowercase. Must use addToTrie.

Preconditions and Postconditions

```
file_name: str -> A string that ends with .txt
Returns: dict -> trie representation of the words in file, all in lowercase
Calls addToTrie()
```

Examples:

```
>>> trie = createDictionaryTrie("list_of_words.txt")
>>> trie
{'b': {'a': {'l': {'l': {'word': True}}, 't': {'s': {'word': True}}}, 'i':
{'r': {'d': {'word': True}}, 'n': {'word': True}}, 'o': {'y': {'word':
True}}}, 't': {'o': {'y': {'s': {'word': True}}}}, 'r': {'e': {'a': {'t':
{'word': True}}, 'e': {'word': True}}}}}
```

Allowed methods, operators, and libraries:

- str.split(sep) returns a list of the words in the string, using *sep* as the delimiter string
- str.lower() returns a string where all characters are lower case
- ord() returns an integer representing the Unicode character
- slicing operator [:] and string concatenation (+)

`wordExists(trie, word)` **4/15 pts**

Given a constructed trie, this function returns a boolean representing whether or not a given word exists in the trie.

Preconditions and Postconditions

```
trie: dict -> trie of words
word: str -> non-empty string
Returns: bool -> True is word is in trie, False otherwise
```

Examples:

```
>>> trie_dict = {'a' : {'word' : True, 'p' : {'p' : {'l' : {'e' : {'word'
: True}}}}, 'i' : {'word' : True}}}
>>> wordExists(trie_dict, 'armor')
False
>>> wordExists(trie_dict, 'apple')
True
>>> wordExists(trie_dict, 'apples')
False
>>> wordExists(trie_dict, 'a')
True
>>> wordExists(trie_dict, 'as')
False
>>> wordExists(trie_dict, 'tot')
False
```

Allowed methods, operators, and libraries:

- slicing operator [:]