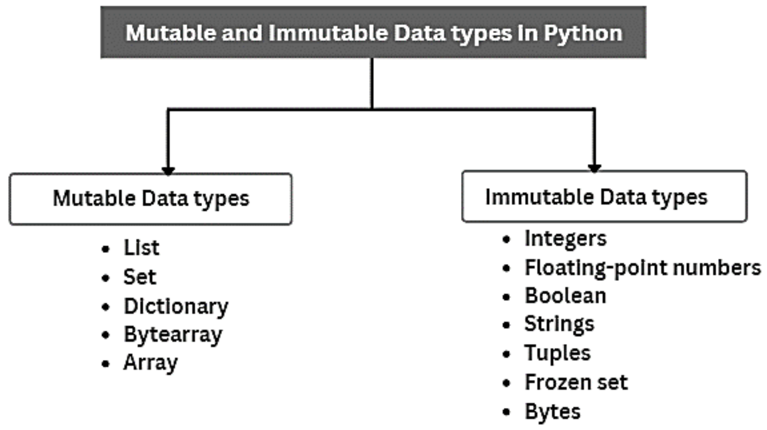
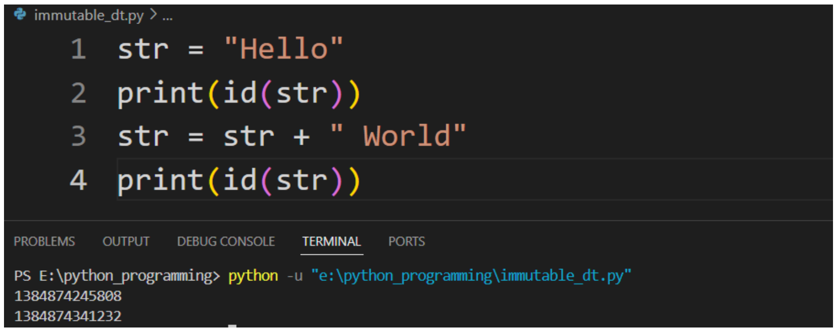
Contents

1. Mutable and Immutable Data Types

Immutable Data Types: Immutable data types are those whose values cannot be modified after they are created. If you try to change an immutable object, a new object is created instead of modifying the existing one.

Mutable Data Types: Mutable data types are those whose values can be changed after they are created.





```python
str = "Hello"
print(id(str))
str = str + " World"
print(id(str))
```

```
PS E:\python_programming> python -u "e:\python_programming\immutable_dt.py"
1384874245808
1384874341232
```

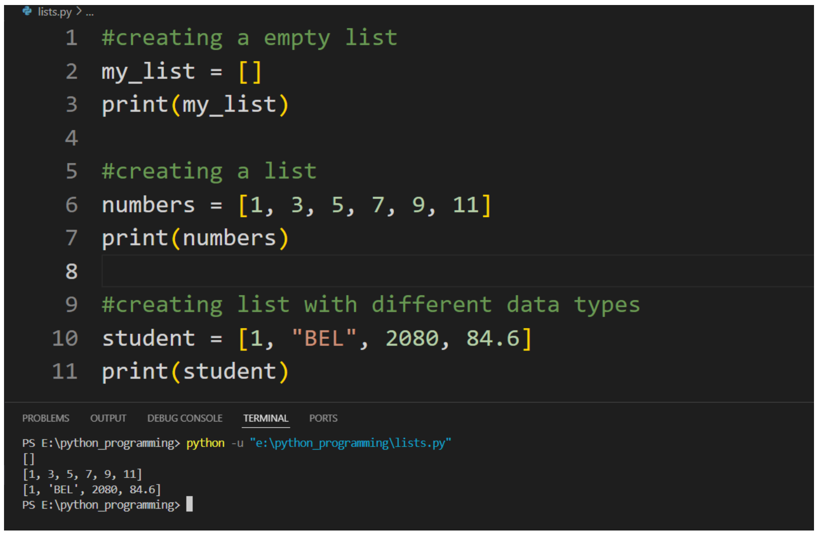
In the given code, we start with a string str initialized to "Hello". When we print the id of str, it gives us the memory address where the string "Hello" is stored. Next, we modify str by concatenating it with " World", resulting in a new string "Hello World". When we print the id of str after this modification, it shows a different memory address. This demonstrates that strings in Python are immutable; modifying a string creates a new string object rather than changing the original one.

2. List and tuple data types.

Lists: A list is an object; like any other object, it can be assigned to a variable. lists allow us to store a sequence of items in a single variable.

Creating a list:

We create a list by placing elements inside the square brackets []. For example: [1, 2, 3, 4] is a list with 4 elements.

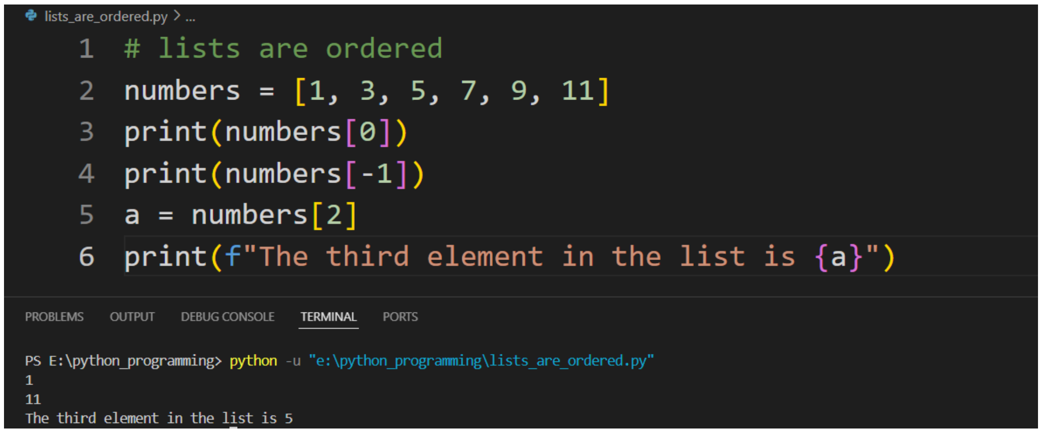


```python
#creating a empty list
my_list = []
print(my_list)

#creating a list
numbers = [1, 3, 5, 7, 9, 11]
print(numbers)

#creating list with different data types
student = [1, "BEL", 2080, 84.6]
print(student)
```

```
PS E:\python_programming> python -u "e:\python_programming\lists.py"
[]
[1, 3, 5, 7, 9, 11]
[1, 'BEL', 2080, 84.6]
PS E:\python_programming>
```

Here, we demonstrate how to create and print different types of lists in Python. First, we create an empty list named my_list using square brackets [] and print it, resulting in an empty list output. Next, we create a list named numbers containing a sequence of integers [1, 3, 5, 7, 9, 11] and print it, which displays the list of numbers as they are. Finally, we create a list named student that includes elements of different data types: an integer, a string, another integer, and a float [1, "BEL", 2080, 84.6], and print this list, showcasing Python's ability to handle lists with mixed data types. The last example shows that lists are **heterogeneous**, but this is prone to error.

The items in a list are **ordered**, and each item has an index indicating its position in the list. The first item in a list is at index 0, the second at index 1, and so on.



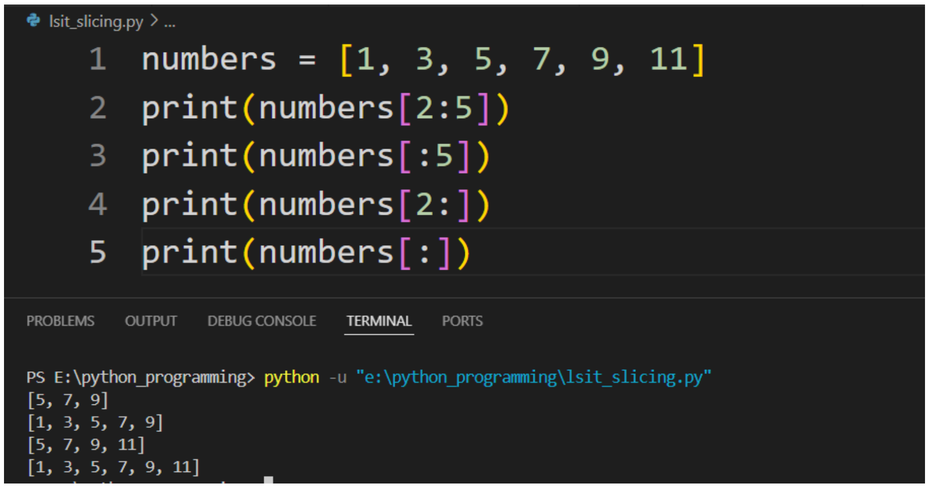
lists_are_ordered.py > ...

```python
# lists are ordered
numbers = [1, 3, 5, 7, 9, 11]
print(numbers[0])
print(numbers[-1])
a = numbers[2]
print(f"The third element in the list is {a}")
```

```
PS E:\python_programming> python -u "e:\python_programming\lists_are_ordered.py"
1
11
The third element in the list is 5
```

Here, we illustrate the ordered nature of lists in Python using the numbers list, which contains the elements [1, 3, 5, 7, 9, 11]. By accessing numbers[0], we retrieve and print the first element of the list, which is 1. Similarly, numbers[-1] allows us to access and print the last element of the list, which is 11. Additionally, we assign the third element of the list, numbers[2], which is 5, to the variable a and print it.

Slicing a list:



lsit_slicing.py > ...

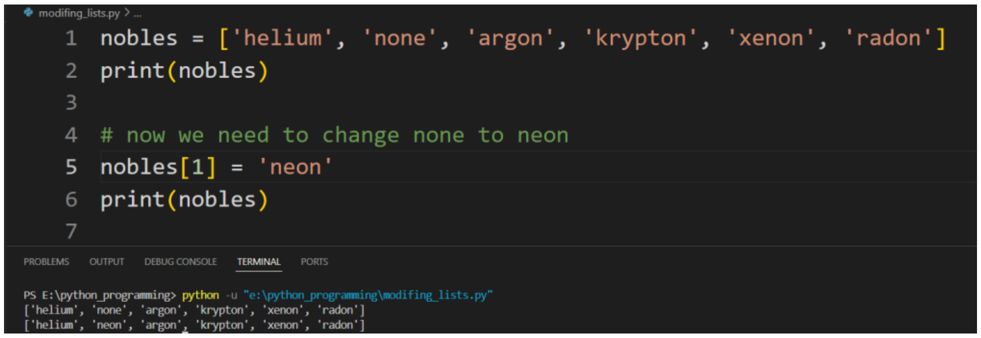
```python
numbers = [1, 3, 5, 7, 9, 11]
print(numbers[2:5])
print(numbers[:5])
print(numbers[2:])
print(numbers[:])
```

```
PS E:\python_programming> python -u "e:\python_programming\lsit_slicing.py"
[5, 7, 9]
[1, 3, 5, 7, 9]
[5, 7, 9, 11]
[1, 3, 5, 7, 9, 11]
```

Here, we demonstrate slicing operations on the numbers list, which contains the elements [1, 3, 5, 7, 9, 11]. By using numbers[2:5], we extract a sublist starting from index 2 up to, but not including, index 5, resulting in [5, 7, 9]. The slice numbers[:5] retrieves elements from the beginning of the list up to, but not including, index 5, producing [1, 3, 5, 7, 9]. Conversely, numbers[2:] gets all elements from index 2 to the end of the list, yielding [5, 7, 9, 11]. Finally, numbers[:] returns a copy of the entire list [1, 3, 5, 7, 9, 11].

Lists are mutable, meaning that their contents can be changed after they are created. This mutability allows for the modification of the list elements, including adding, removing, or changing elements, without creating a new list.

Suppose you're typing in a list of the noble gases.

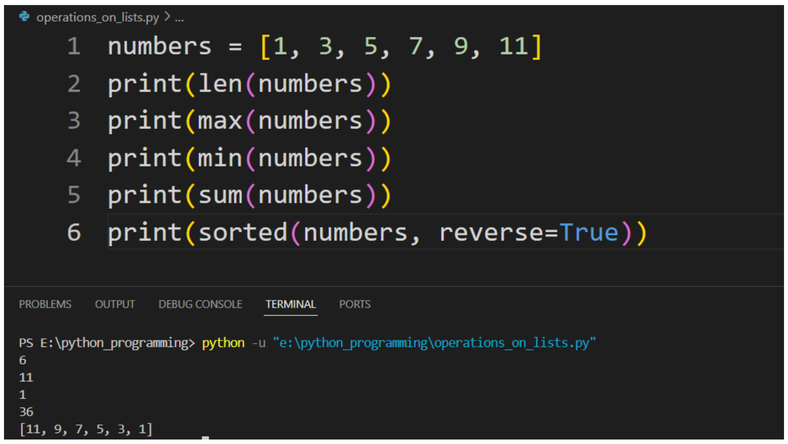


```python
nobles = ['helium', 'none', 'argon', 'krypton', 'xenon', 'radon']
print(nobles)

# now we need to change none to neon
nobles[1] = 'neon'
print(nobles)
```

```
PS E:\python_programming> python -u "e:\python_programming\modifing_lists.py"
['helium', 'none', 'argon', 'krypton', 'xenon', 'radon']
['helium', 'neon', 'argon', 'krypton', 'xenon', 'radon']
```

In the code, nobles[1] was used on the left side of the assignment operator. It can also be used on the right side. In general, an expression of the form L[i] (list L at index i) behaves just like a simple variable. if L[i] is on the left of an assignment statement it means "Look up the memory address at index i of list L so it can be overwritten". In contrast to lists, numbers and strings are immutable. You cannot, for example, change a letter in a string.

Operations on the list:

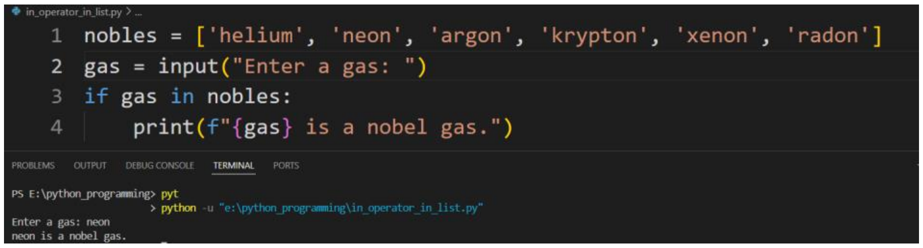


```python
numbers = [1, 3, 5, 7, 9, 11]
print(len(numbers))
print(max(numbers))
print(min(numbers))
print(sum(numbers))
print(sorted(numbers, reverse=True))
```

```
PS E:\python_programming> python -u "e:\python_programming\operations_on_lists.py"
6
11
1
36
[11, 9, 7, 5, 3, 1]
```

Here, we demonstrate various built-in functions that operate on the numbers list, which contains the elements [1, 3, 5, 7, 9, 11]. First, we use len(numbers) to determine and print the length of the list, which is 6. Next, we apply max(numbers) to find and print the maximum value in the list, which is 11, and min(numbers) to find and print the minimum value, which is 1. We then use sum(numbers) to calculate and print the sum of all elements in the list, resulting in 36. Finally, we call sorted(numbers, reverse=True) to sort the list in descending order and print the sorted list, yielding [11, 9, 7, 5, 3, 1].

The in operator in list

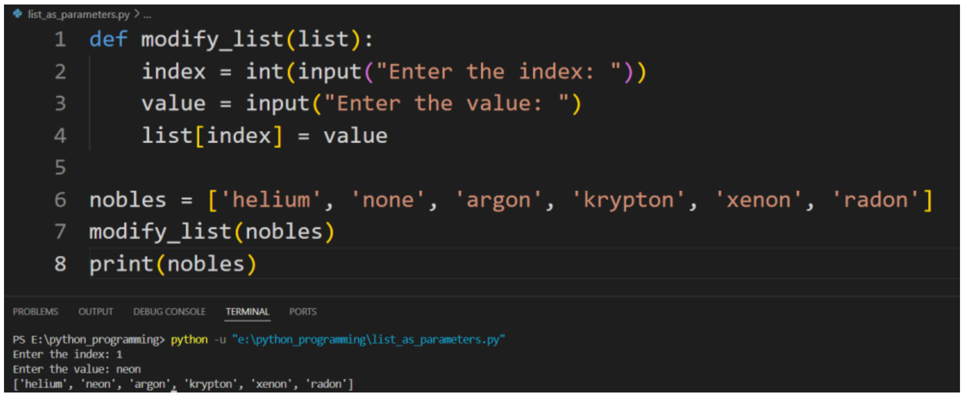


```python
nobles = ['helium', 'neon', 'argon', 'krypton', 'xenon', 'radon']
gas = input("Enter a gas: ")
if gas in nobles:
    print(f"{gas} is a nobel gas.")
```

```
PS E:\python_programming> pyt
                          > python -u "e:\python_programming\in_operator_in_list.py"
Enter a gas: neon
neon is a nobel gas.
```

In this code, we have a list named nobles that contains the names of noble gases: ['helium', 'neon', 'argon', 'krypton', 'xenon', 'radon']. The if gas in nobles: statement checks whether the entered gas is present in the nobles list. If the condition is true, meaning the input gas is indeed a noble gas, the program prints a confirmation message in the format "{gas} is a noble gas."

Aliasing: An alias is an alternative name for something. Aliasing occurs when you use list parameters. Aliasing is one of the reasons why the notion of mutability is important.

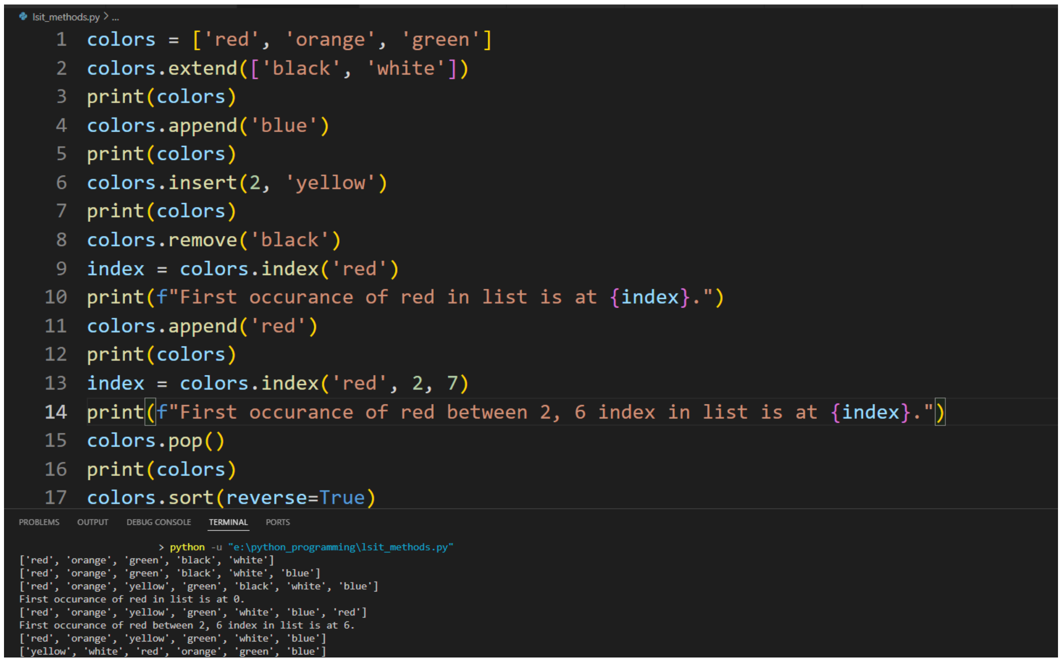


```python
def modify_list(list):
    index = int(input("Enter the index: "))
    value = input("Enter the value: ")
    list[index] = value

nobles = ['helium', 'none', 'argon', 'krypton', 'xenon', 'radon']
modify_list(nobles)
print(nobles)
```

```
PS E:\python_programming> python -u "e:\python_programming\list_as_parameters.py"
Enter the index: 1
Enter the value: neon
['helium', 'neon', 'argon', 'krypton', 'xenon', 'radon']
```

In this code, we define a function modify_list(list) that modifies an element of a list based on user input. The function first prompts the user to enter an index and then a value. It then updates the list at the specified index with the given value. The list nobles are initialized with the elements ['helium', 'none', 'argon', 'krypton', 'xenon', 'radon']. When the modify_list(nobles) function is called, the nobles list is passed as an argument. Due to list aliasing, the list parameter in the function refers to the same memory location as nobles, meaning any modifications made to list within the function will directly affect nobles. After the function modifies the list based on user input, the changes are reflected in nobles, and the updated list is printed.

Lists Methods:



lsit_methods.py > ...

```python
colors = ['red', 'orange', 'green']
colors.extend(['black', 'white'])
print(colors)
colors.append('blue')
print(colors)
colors.insert(2, 'yellow')
print(colors)
colors.remove('black')
index = colors.index('red')
print(f"First occurance of red in list is at {index}.")
colors.append('red')
print(colors)
index = colors.index('red', 2, 7)
print(f"First occurance of red between 2, 6 index in list is at {index}.")
colors.pop()
print(colors)
colors.sort(reverse=True)
```

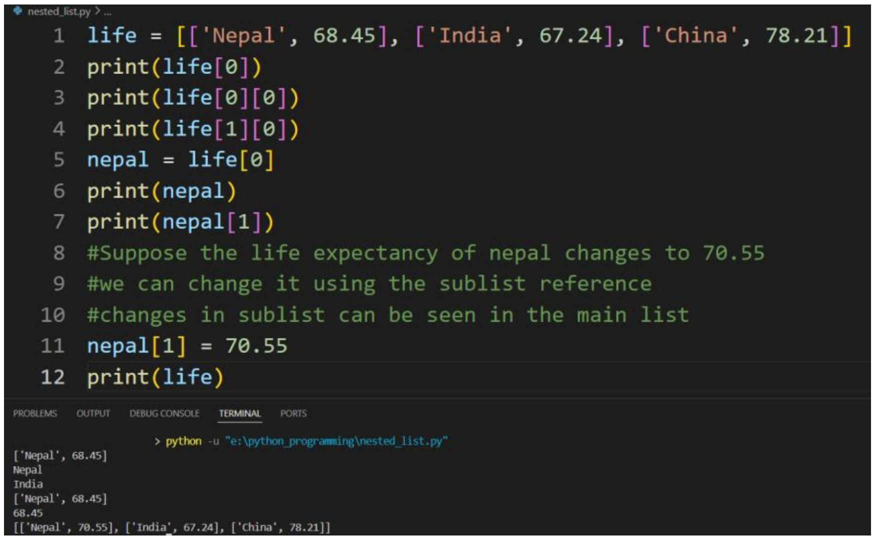
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
> python -u "e:\python_programming\lsit_methods.py"
['red', 'orange', 'green', 'black', 'white']
['red', 'orange', 'green', 'black', 'white', 'blue']
['red', 'orange', 'yellow', 'green', 'black', 'white', 'blue']
First occurance of red in list is at 0.
['red', 'orange', 'yellow', 'green', 'white', 'blue', 'red']
First occurance of red between 2, 6 index in list is at 6.
['red', 'orange', 'yellow', 'green', 'white', 'blue']
['yellow', 'white', 'red', 'orange', 'green', 'blue']
```

In this code, we start with a list colors containing ['red', 'orange', 'green']. The extend method is used to add multiple elements ['black', 'white'] to the end of the list, resulting in ['red', 'orange', 'green', 'black', 'white']. Next, the append method adds 'blue' to the end, giving ['red', 'orange', 'green', 'black', 'white', 'blue']. The insert method places 'yellow' at index 2, resulting in ['red', 'orange', 'yellow', 'green', 'black', 'white', 'blue']. The remove method deletes 'black', so the list becomes ['red', 'orange', 'yellow', 'green', 'white', 'blue']. The index method finds the first occurrence of 'red' at index 0. After appending another 'red', the list is ['red', 'orange', 'yellow', 'green', 'white', 'blue', 'red']. The index method is used again to find the first occurrence of 'red' between indices 2 and 7, which is at index 6. The pop method removes the last element, giving ['red', 'orange', 'yellow', 'green', 'white', 'blue']. Finally, the sort method sorts the list in reverse order, resulting in ['yellow', 'white', 'red', 'orange', 'green', 'blue'].

Working with a List of Lists

We said in Lists Are Heterogeneous, that lists can contain any type of data. That means that they can contain other lists. A list whose items are lists is called a nested list. For example, the following nested list describes life expectancies in different countries.

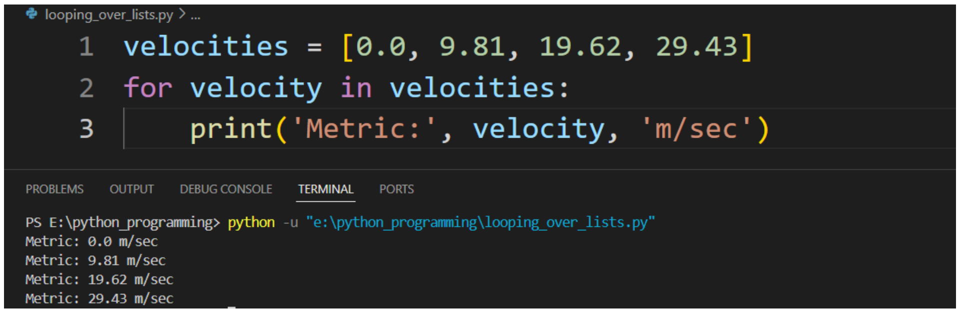


```python
life = [['Nepal', 68.45], ['India', 67.24], ['China', 78.21]]
print(life[0])
print(life[0][0])
print(life[1][0])
nepal = life[0]
print(nepal)
print(nepal[1])
#Suppose the life expectancy of nepal changes to 70.55
#we can change it using the sublist reference
#changes in sublist can be seen in the main list
nepal[1] = 70.55
print(life)
```

```
> python -u "e:\python_programming\nested_list.py"
['Nepal', 68.45]
Nepal
India
['Nepal', 68.45]
68.45
[['Nepal', 70.55], ['India', 67.24], ['China', 78.21]]
```

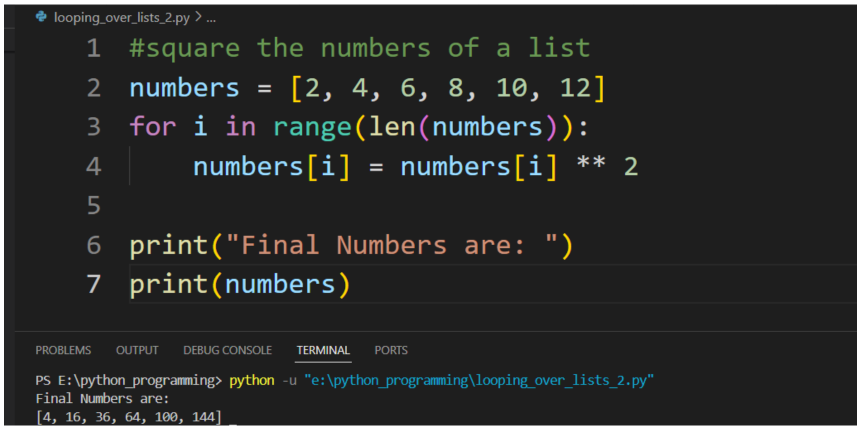
In this code, we have a nested list life containing sublists that represent countries and their respective life expectancies: [['Nepal', 68.45], ['India', 67.24], ['China', 78.21]]. The print(life[0]) statement outputs the first sublist ['Nepal', 68.45], and print(life[0][0]) prints the first element of the first sublist, which is 'Nepal'. Similarly, print(life[1][0]) prints 'India', the first element of the second sublist. We then assign the first sublist of life to the variable nepal and print it, resulting in ['Nepal', 68.45]. Printing nepal[1] displays 68.45, the life expectancy of Nepal. To update the life expectancy of Nepal to 70.55, we change the second element of the nepal sublist. Since nepal is a reference to the first sublist of life, modifying nepal directly updates life. Thus, when we print life again, it shows the updated list: [['Nepal', 70.55], ['India', 67.24], ['China', 78.21]].

Looping over lists:



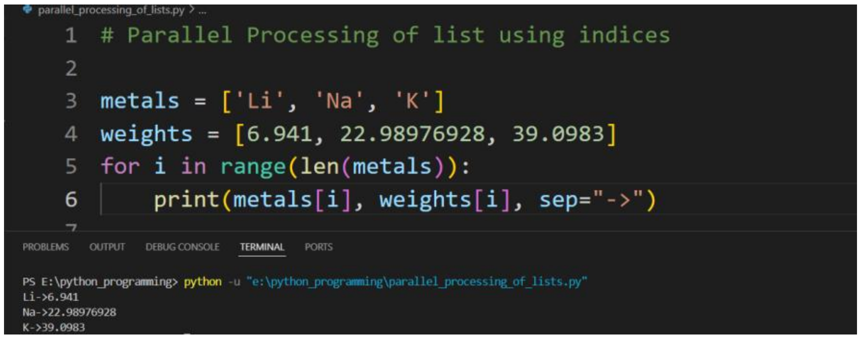
```python
velocities = [0.0, 9.81, 19.62, 29.43]
for velocity in velocities:
    print('Metric:', velocity, 'm/sec')
```

```
PS E:\python_programming> python -u "e:\python_programming\looping_over_lists.py"
Metric: 0.0 m/sec
Metric: 9.81 m/sec
Metric: 19.62 m/sec
Metric: 29.43 m/sec
```



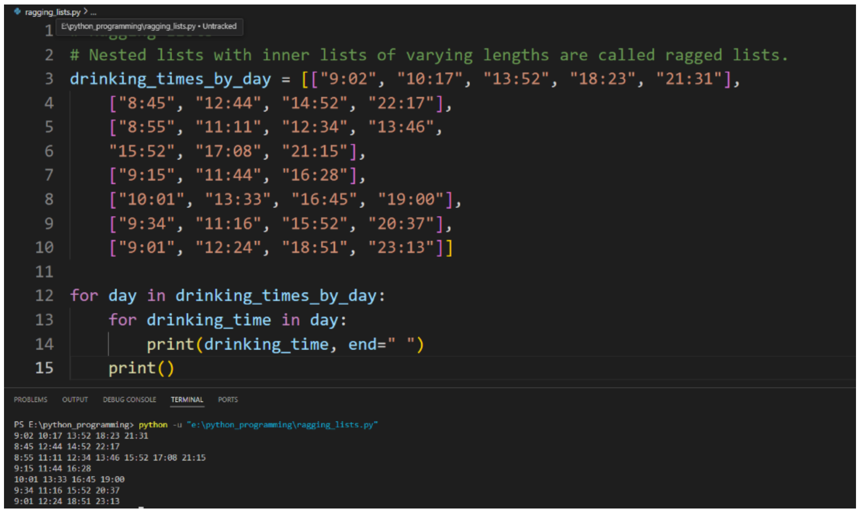
```python
#square the numbers of a list
numbers = [2, 4, 6, 8, 10, 12]
for i in range(len(numbers)):
    numbers[i] = numbers[i] ** 2

print("Final Numbers are: ")
print(numbers)
```

```
PS E:\python_programming> python -u "e:\python_programming\looping_over_lists_2.py"
Final Numbers are:
[4, 16, 36, 64, 100, 144]
```



```python
# Parallel Processing of list using indices

metals = ['Li', 'Na', 'K']
weights = [6.941, 22.98976928, 39.0983]
for i in range(len(metals)):
    print(metals[i], weights[i], sep="->")
```

```
PS E:\python_programming> python -u "e:\python_programming\parallel_processing_of_lists.py"
Li->6.941
Na->22.98976928
K->39.0983
```

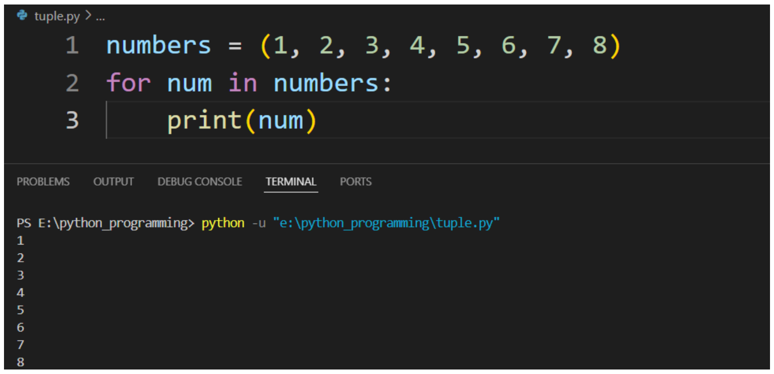


```python
# Nested lists with inner lists of varying lengths are called ragged lists.
drinking_times_by_day = [["9:02", "10:17", "13:52", "18:23", "21:31"],
    ["8:45", "12:44", "14:52", "22:17"],
    ["8:55", "11:11", "12:34", "13:46",
    "15:52", "17:08", "21:15"],
    ["9:15", "11:44", "16:28"],
    ["10:01", "13:33", "16:45", "19:00"],
    ["9:34", "11:16", "15:52", "20:37"],
    ["9:01", "12:24", "18:51", "23:13"]]

for day in drinking_times_by_day:
    for drinking_time in day:
        print(drinking_time, end=" ")
    print()
```

```
PS E:\python_programming> python -u "e:\python_programming\ragging_lists.py"
9:02 10:17 13:52 18:23 21:31
8:45 12:44 14:52 22:17
8:55 11:11 12:34 13:46 15:52 17:08 21:15
9:15 11:44 16:28
10:01 13:33 16:45 19:00
9:34 11:16 15:52 20:37
9:01 12:24 18:51 23:13
```

Tuple Data type:

Python also has an immutable sequence type called a tuple. Tuples are written using parentheses instead of brackets; like strings and lists, they can be subscripted, sliced, and looped over.



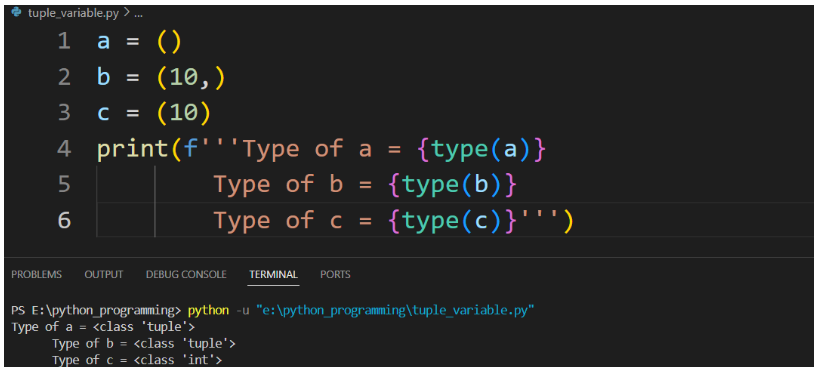
tuple.py > ...

```python
numbers = (1, 2, 3, 4, 5, 6, 7, 8)
for num in numbers:
    print(num)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS E:\python_programming> python -u "e:\python_programming\tuple.py"
1
2
3
4
5
6
7
8
```

In this code, we define a tuple named numbers containing the elements (1, 2, 3, 4, 5, 6, 7, 8). We then use a for loop to iterate over each element in the numbers tuple. In each iteration, the current element (assigned to the variable num) is printed to the console.



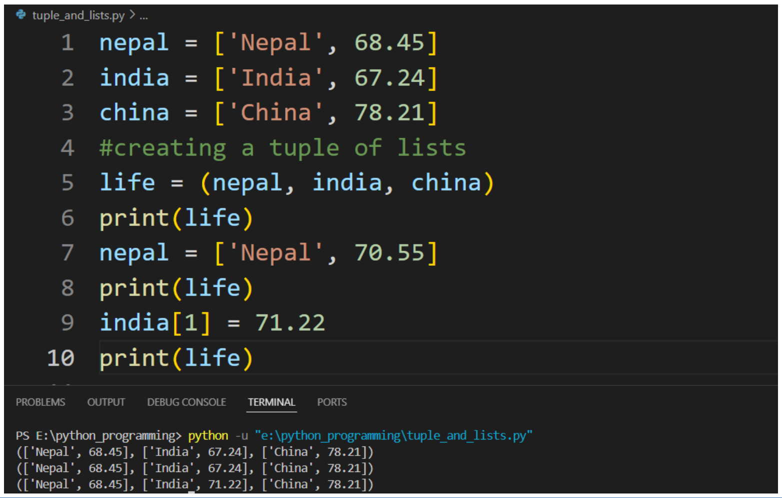
tuple_variable.py > ...

```python
a = ()
b = (10,)
c = (10)
print(f'''Type of a = {type(a)}
        Type of b = {type(b)}
        Type of c = {type(c)}''')
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS E:\python_programming> python -u "e:\python_programming\tuple_variable.py"
Type of a = <class 'tuple'>
        Type of b = <class 'tuple'>
        Type of c = <class 'int'>
```

In the provided code, the variable a is an empty tuple, indicated by (), resulting in type(a) being <class 'tuple'>. The variable b is defined as a tuple with one element, written as (10,), where the comma signifies it's a tuple, hence type(b) is <class 'tuple'>. However, c is assigned the value (10) without a trailing comma, which is interpreted as an integer rather than a tuple, resulting in type(c) being <class 'int'>.



tuple_and_lists.py > ...

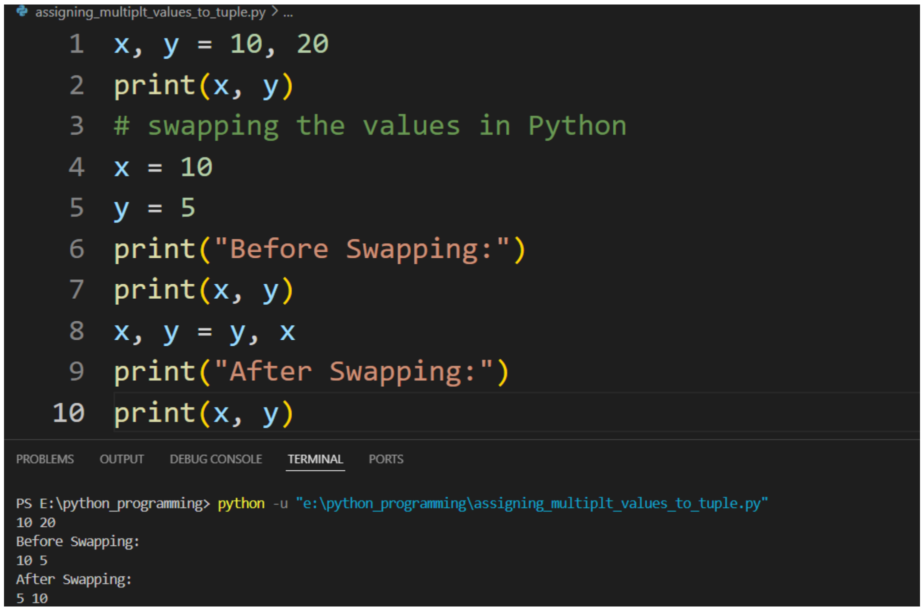
```python
nepal = ['Nepal', 68.45]
india = ['India', 67.24]
china = ['China', 78.21]
#creating a tuple of lists
life = (nepal, india, china)
print(life)
nepal = ['Nepal', 70.55]
print(life)
india[1] = 71.22
print(life)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS E:\python_programming> python -u "e:\python_programming\tuple_and_lists.py"
(['Nepal', 68.45], ['India', 67.24], ['China', 78.21])
(['Nepal', 68.45], ['India', 67.24], ['China', 78.21])
(['Nepal', 68.45], ['India', 71.22], ['China', 78.21])
```

The code creates a tuple named life that contains three lists, each representing a country and its life expectancy: Nepal (68.45), India (67.24), and China (78.21). When the life tuple is printed initially, it displays these values. Next, the list nepal is reassigned to a new list with updated values (Nepal, 70.55), but since tuples store references to objects and not the objects themselves, the original reference in the life tuple remains unchanged. When india[1] is updated to 71.22, this change is reflected in the life tuple because lists are mutable and the tuple still references the original india list. Thus, the final print statement shows the updated life expectancy for India within the life tuple but not for Nepal.

Assigning to Multiple Variables Using Tuples:



```python
x, y = 10, 20
print(x, y)
# swapping the values in Python
x = 10
y = 5
print("Before Swapping:")
print(x, y)
x, y = y, x
print("After Swapping:")
print(x, y)
```

```
PS E:\python_programming> python -u "e:\python_programming\assigning_multiplt_values_to_tuple.py"
10 20
Before Swapping:
10 5
After Swapping:
5 10
```

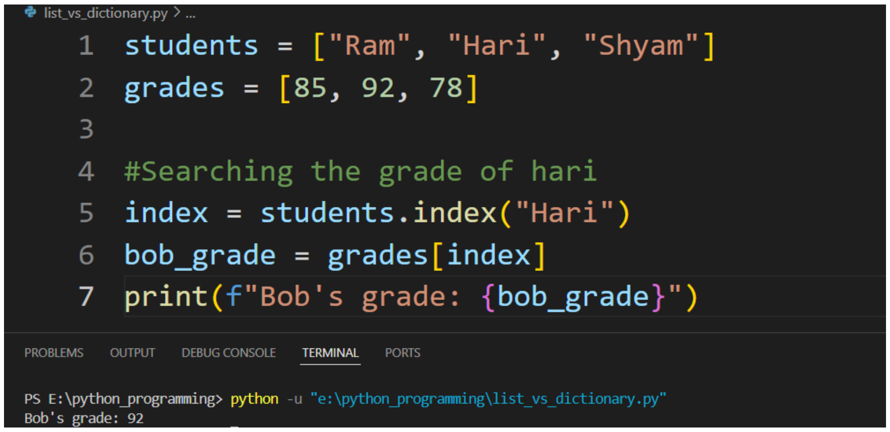
In the provided code, x, y = 10, 20 assigns the values 10 and 20 to x and y respectively, and the print(x, y) statement outputs "10 20". Next, x is assigned 10 and y is assigned 5, and the values are printed as "10 5" before swapping. The line x, y = y, x performs a swap by packing the values of y and x into a tuple (y, x) and then unpacking them back into x and y, effectively exchanging their values. After this swap, x holds the value 5 and y holds the value 10, and the final print statement outputs "5 10". This mechanism of tuple packing and unpacking allows for a concise and efficient way to swap variable values in Python.

3. Dictionary data types

Also known as a map, a dictionary is an unordered mutable collection of key/value pairs. In plain English, Python's dictionaries are like dictionaries that map words to definitions. They associate a key (like a word) with a value (such as a definition). The keys form a set: any particular key can appear once at most in a dictionary. In dictionary, keys must be immutable (though the values associated with them don't have to be).

Storing and Accessing Student Grades

Using Lists:



```python
students = ["Ram", "Hari", "Shyam"]
grades = [85, 92, 78]

#Searching the grade of hari
index = students.index("Hari")
bob_grade = grades[index]
print(f"Bob's grade: {bob_grade}")
```

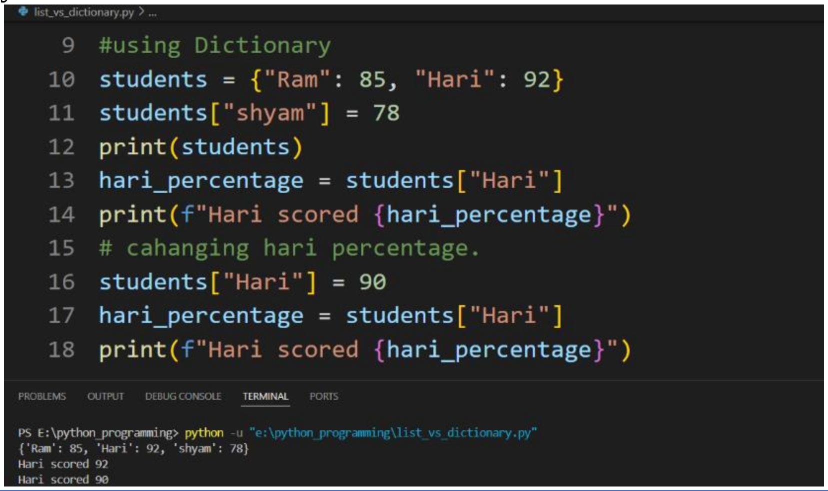
```
PS E:\python_programming> python -u "e:\python_programming\list_vs_dictionary.py"
Bob's grade: 92
```

Dictionaries are created by putting key/value pairs inside braces (each key is followed by a colon and then by its value):

Example: variable = {key: value, key: value}

To get the value associated with a key, we put the key in square brackets, much like indexing into a list: variable[key]

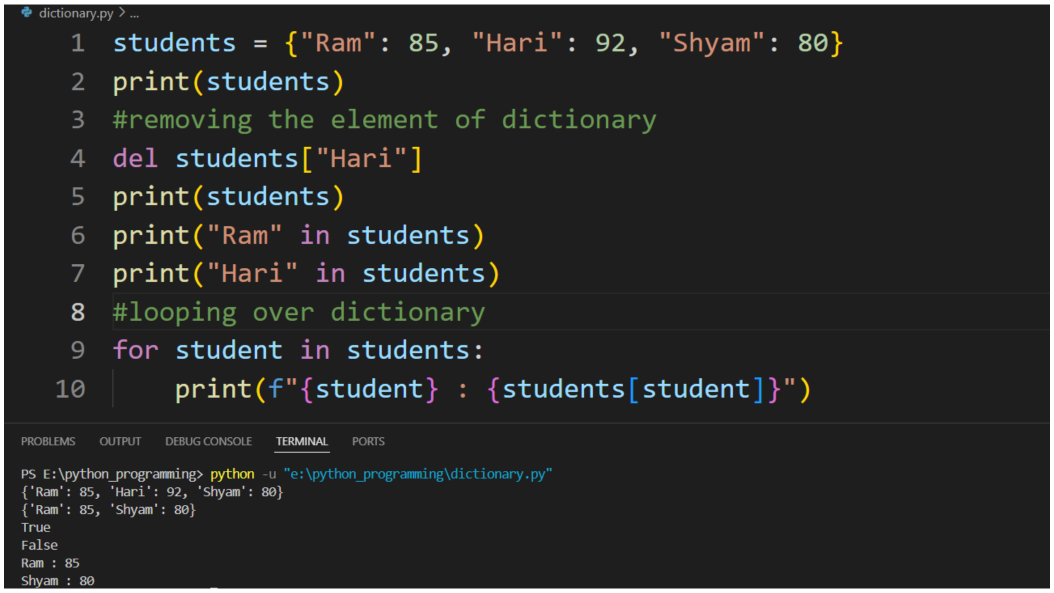
The empty dictionary is written {}



```python
#using Dictionary
students = {"Ram": 85, "Hari": 92}
students["shyam"] = 78
print(students)
hari_percentage = students["Hari"]
print(f"Hari scored {hari_percentage}")
# cahanging hari percentage.
students["Hari"] = 90
hari_percentage = students["Hari"]
print(f"Hari scored {hari_percentage}")
```

```
PS E:\python_programming> python -u "e:\python_programming\list_vs_dictionary.py"
{'Ram': 85, 'Hari': 92, 'shyam': 78}
Hari scored 92
Hari scored 90
```

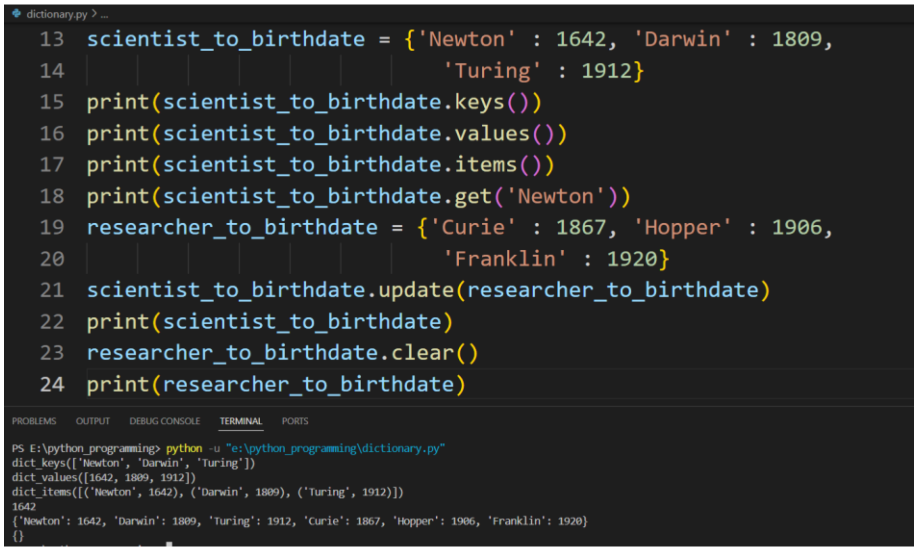
The code first demonstrates how to search for a student's grade using a list. It defines two lists: students with names and grades with corresponding grades. To find Hari's grade, it locates Hari's index in the students list and uses that index to get the grade from the grades list. This method is commented out. Then, the code demonstrates using a dictionary for the same task. It initializes a dictionary students with names as keys and grades as values, adds Shyam's grade to the dictionary, and prints the entire dictionary. It retrieves and prints Hari's grade using the key "Hari". The code then updates Hari's grade to 90 in the dictionary and prints the updated grade.
This illustrates that dictionaries provide a more efficient and readable way to access, update, and manage student grades compared to lists.



```python
students = {"Ram": 85, "Hari": 92, "Shyam": 80}
print(students)
#removing the element of dictionary
del students["Hari"]
print(students)
print("Ram" in students)
print("Hari" in students)
#looping over dictionary
for student in students:
    print(f"{student} : {students[student]}")
```

```
PS E:\python_programming> python -u "e:\python_programming\dictionary.py"
{'Ram': 85, 'Hari': 92, 'Shyam': 80}
{'Ram': 85, 'Shyam': 80}
True
False
Ram : 85
Shyam : 80
```

The code starts by creating a dictionary students with names as keys and grades as values, then prints the dictionary. It removes the entry for "Hari" using the del statement and prints the updated dictionary, which now only contains "Ram" and "Shyam". The code then checks if "Ram" and "Hari" are still in the dictionary, printing True for "Ram" and False for "Hari". Finally, it loops over the dictionary, printing each student's name and their corresponding grade. This example demonstrates how to create, modify, and iterate over a dictionary in Python, highlighting the ease of key-based data management.



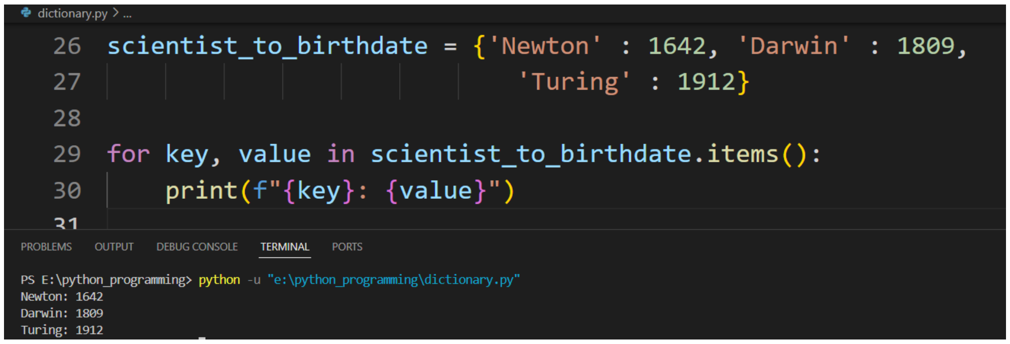
dictionary.py > ...

```python
scientist_to_birthdate = {'Newton' : 1642, 'Darwin' : 1809,
                          'Turing' : 1912}
print(scientist_to_birthdate.keys())
print(scientist_to_birthdate.values())
print(scientist_to_birthdate.items())
print(scientist_to_birthdate.get('Newton'))
researcher_to_birthdate = {'Curie' : 1867, 'Hopper' : 1906,
                           'Franklin' : 1920}
scientist_to_birthdate.update(researcher_to_birthdate)
print(scientist_to_birthdate)
researcher_to_birthdate.clear()
print(researcher_to_birthdate)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS E:\python_programming> python -u "e:\python_programming\dictionary.py"
dict_keys(['Newton', 'Darwin', 'Turing'])
dict_values([1642, 1809, 1912])
dict_items([('Newton', 1642), ('Darwin', 1809), ('Turing', 1912)])
1642
{'Newton': 1642, 'Darwin': 1809, 'Turing': 1912, 'Curie': 1867, 'Hopper': 1906, 'Franklin': 1920}
{}
```

A dictionary scientist_to_birthdate is created, mapping scientists to their birth years. The code prints the dictionary's keys, values, and items, retrieves Newton's birth year using get, and then merges another dictionary researcher_to_birthdate into scientist_to_birthdate using the update method. Finally, it clears the researcher_to_birthdate dictionary.



dictionary.py > ...

```python
scientist_to_birthdate = {'Newton' : 1642, 'Darwin' : 1809,
                          'Turing' : 1912}

for key, value in scientist_to_birthdate.items():
    print(f"{key}: {value}")
```

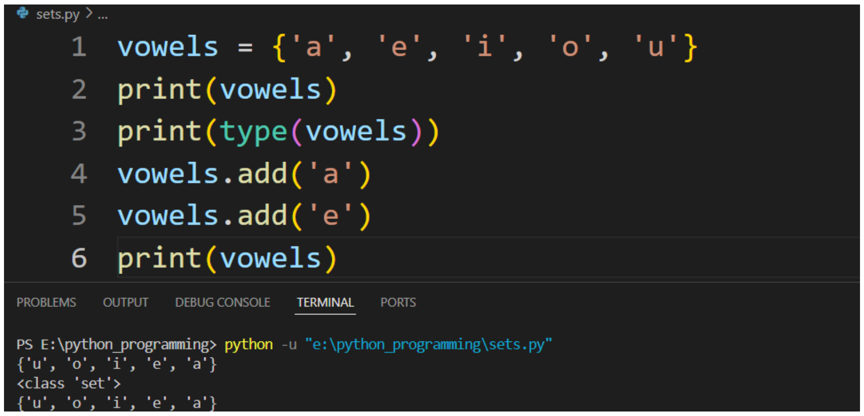
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS E:\python_programming> python -u "e:\python_programming\dictionary.py"
Newton: 1642
Darwin: 1809
Turing: 1912
```

4. Sets data types

A set is an unordered collection of distinct items. Unordered means that items aren’t stored in any particular order. Something is either in the set or it’s not, but there’s no notion of it being the first, second, or last item. Distinct means that any item appears in a set at most once; in other words, there are no duplicates. Python has a type called set that allows us to store mutable collections of unordered, distinct items. (Remember that a mutable object is one that you can modify.)

Here we create a set containing the vowels:



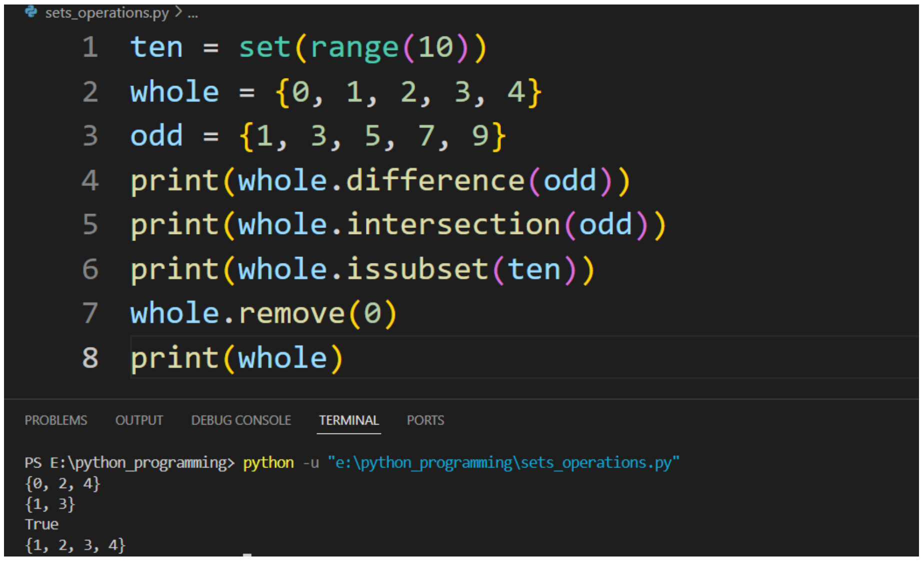
sets.py > ...

```python
vowels = {'a', 'e', 'i', 'o', 'u'}
print(vowels)
print(type(vowels))
vowels.add('a')
vowels.add('e')
print(vowels)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS E:\python_programming> python -u "e:\python_programming\sets.py"
{'u', 'o', 'i', 'e', 'a'}
<class 'set'>
{'u', 'o', 'i', 'e', 'a'}
```

The code demonstrates the use of a set in Python to store vowel characters. A set named vowels is initialized with the characters 'a', 'e', 'i', 'o', and 'u'. When print(vowels) is executed, it displays the contents of the set, and print(type(vowels)) confirms that vowels is of type set. The add method is then used to attempt to add the characters 'a' and 'e' again to the set. However, since sets do not allow duplicate elements, the set remains unchanged.



sets_operations.py > ...

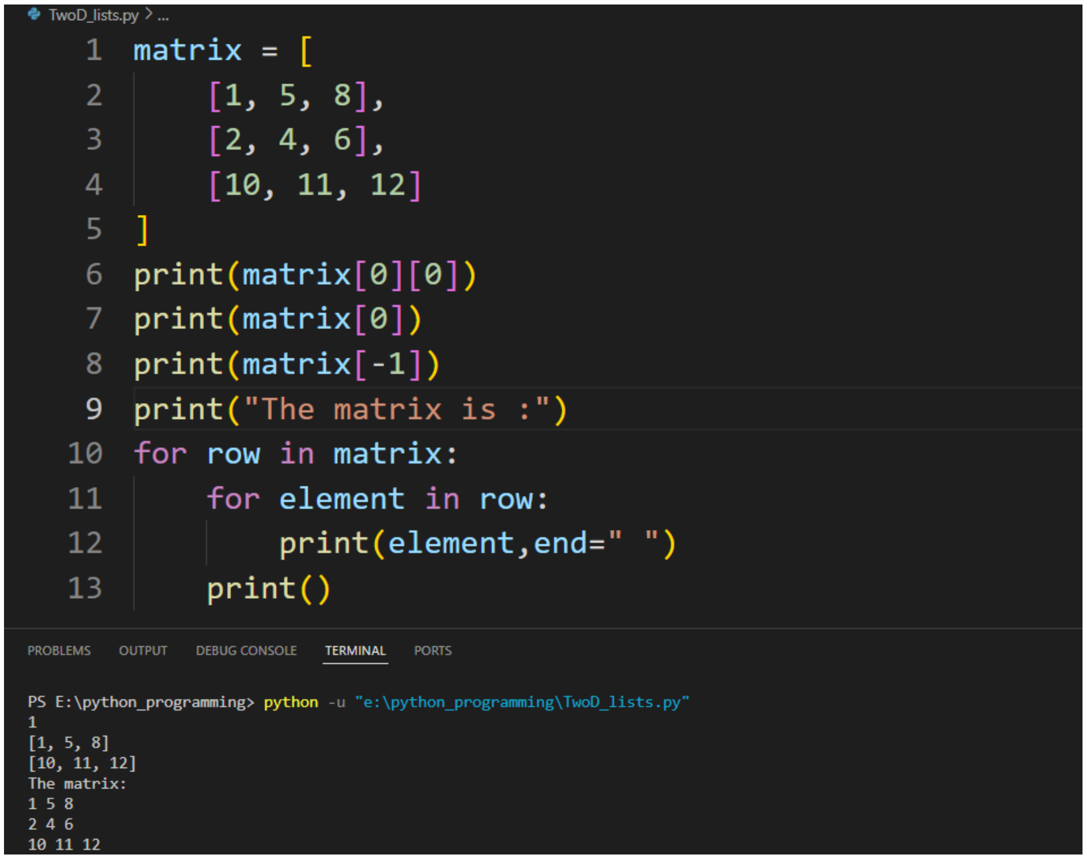
```python
ten = set(range(10))
whole = {0, 1, 2, 3, 4}
odd = {1, 3, 5, 7, 9}
print(whole.difference(odd))
print(whole.intersection(odd))
print(whole.issubset(ten))
whole.remove(0)
print(whole)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS E:\python_programming> python -u "e:\python_programming\sets_operations.py"
{0, 2, 4}
{1, 3}
True
{1, 2, 3, 4}
```

The code demonstrates various set operations in Python. It first creates a set ten containing the numbers from 0 to 9 using the range function. It also defines two more sets: whole containing the numbers 0 through 4, and odd containing the odd numbers 1, 3, 5, 7, and 9. The difference method is used to find elements in whole that are not in odd, resulting in {0, 2, 4}. The intersection method finds elements common to both whole and odd, resulting in {1, 3}. The issubset method checks if all elements of whole are in ten, returning True since every element in whole is indeed in ten. The remove method removes the element 0 from whole, updating it to {1, 2, 3, 4}.

5. Two dimensional Lists



```python
matrix = [
    [1, 5, 8],
    [2, 4, 6],
    [10, 11, 12]
]
print(matrix[0][0])
print(matrix[0])
print(matrix[-1])
print("The matrix is :")
for row in matrix:
    for element in row:
        print(element,end=" ")
    print()
```

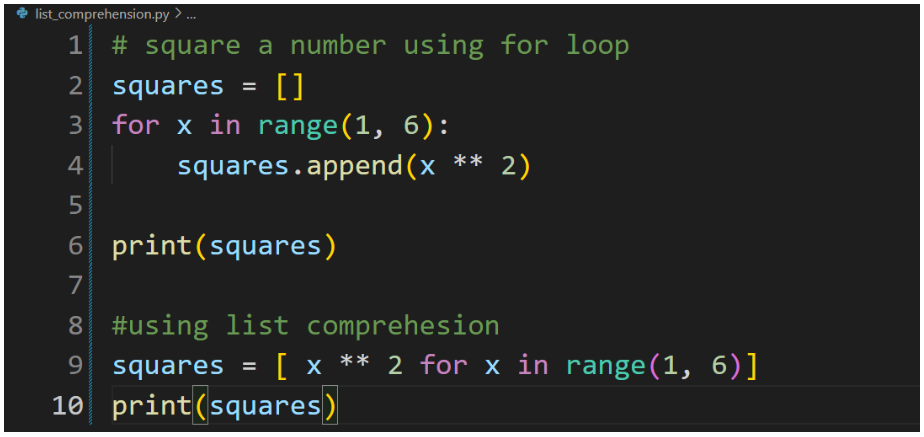
```
PS E:\python_programming> python -u "e:\python_programming\TwoD_lists.py"
1
[1, 5, 8]
[10, 11, 12]
The matrix:
1 5 8
2 4 6
10 11 12
```

In the provided code, a 2D list matrix is defined with three rows and three columns. The element at the first row and first column is accessed and printed using matrix[0][0], which outputs 1. The entire first row is printed using matrix[0], outputting [1, 5, 8], and the last row is printed using matrix[-1], outputting [10, 11, 12]. The code then prints "The matrix is :" followed by iterating through each row of the matrix and each element within those rows, printing the elements in a row-wise manner. The elements are printed in a single line for each row, resulting in the output: "1 5 8", "2 4 6", and "10 11 12" on separate lines, thereby displaying the matrix in a structured format.

6. List Comprehensions

ist comprehensions are a concise and powerful way to create lists in Python. They provide a more compact syntax compared to traditional loops and can often make your code more readable. List comprehensions consist of an expression followed by a for clause, then zero or more for or if clauses. The result will be a new list resulting from evaluating the expression in the context of the for and if clauses which follow it. Here's the general syntax:

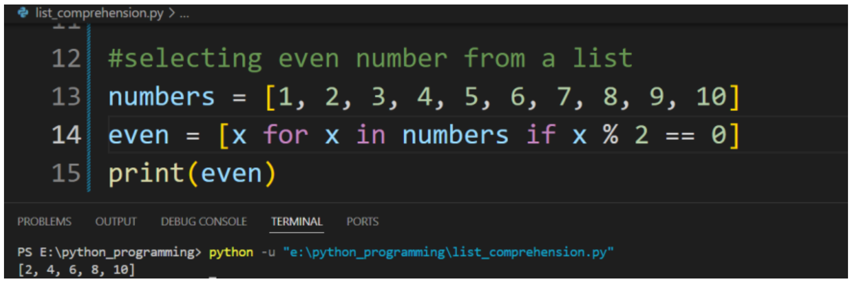
[expression for item in iterable if condition]



list_comprehension.py > ...

```python
# square a number using for loop
squares = []
for x in range(1, 6):
    squares.append(x ** 2)

print(squares)

#using list comprehension
squares = [ x ** 2 for x in range(1, 6)]
print(squares)
```

Both the for loop and list comprehension achieve the same task of squaring numbers from 1 to 5 and storing the results in a list named squares. In the for loop approach, a list named squares is initialized, then each number from 1 to 5 is squared using the ** operator within the loop, and the result is appended to the list squares using the append() method. On the other hand, the list comprehension approach achieves the same result in a more concise and readable manner. It directly generates the list squares by iterating over the range from 1 to 6 (exclusive) and squaring each number x using the expression x ** 2. The resulting list comprehension [x ** 2 for x in range(1, 6)] is equivalent to the for loop in functionality but provides a more compact syntax. Both approaches produce the same output: [1, 4, 9, 16, 25], representing the squares of numbers from 1 to 5.



list_comprehension.py > ...

```python
#selecting even number from a list
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even = [x for x in numbers if x % 2 == 0]
print(even)
```

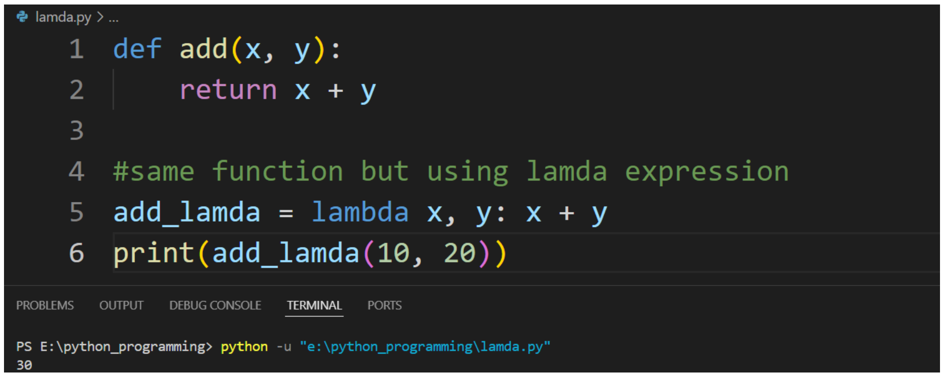
```
PS E:\python_programming> python -u "e:\python_programming\list_comprehension.py"
[2, 4, 6, 8, 10]
```

7. Lambda (Anonymous Function)

A lambda expression in Python is a way to create anonymous functions. Unlike regular functions defined using the def keyword, lambda functions are small, one-liner functions that can be defined without a name. They are often used in situations where a small function is needed for a short period of time, such as for sorting, filtering, or mapping data.

The general syntax of a lambda expression is:

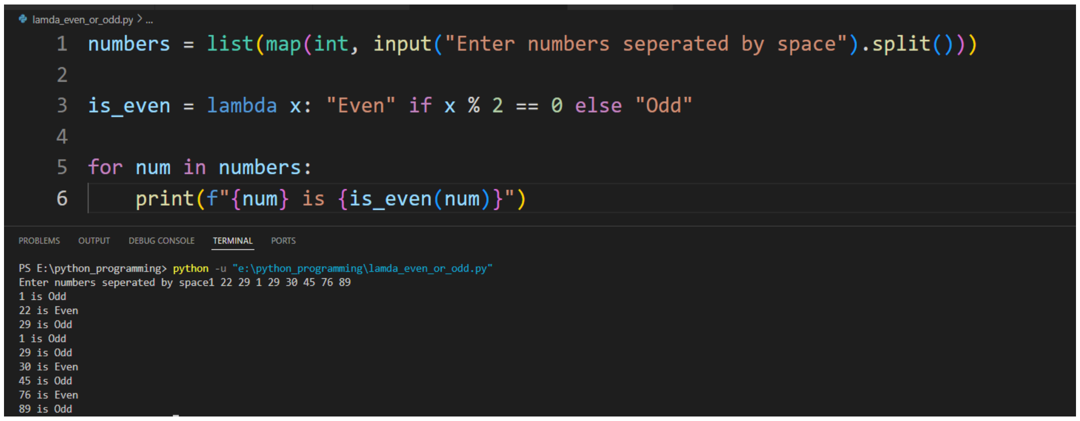
lambda arguments: expression



```python
def add(x, y):
    return x + y

#same function but using lamda expression
add_lamda = lambda x, y: x + y
print(add_lamda(10, 20))
```

```
PS E:\python_programming> python -u "e:\python_programming\lamda.py"
30
```

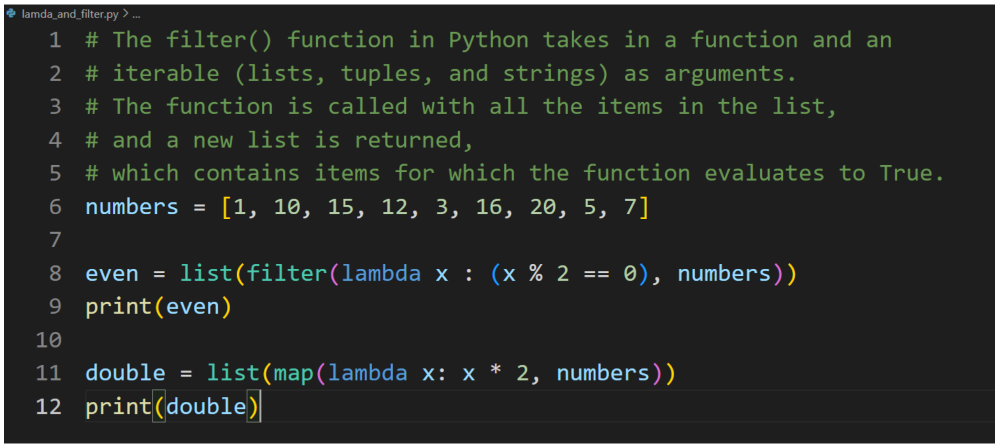
The code defines a function add using the def keyword, which takes two arguments, x and y, and returns their sum. It then defines an equivalent function using a lambda expression, add_lambda, which also takes two arguments, x and y, and returns their sum in a more concise manner. The lambda expression lambda x, y: x + y achieves the same result as the add function but without explicitly naming the function. When add_lambda(10, 20) is called, it returns 30, demonstrating that the lambda function works just like the regular function.



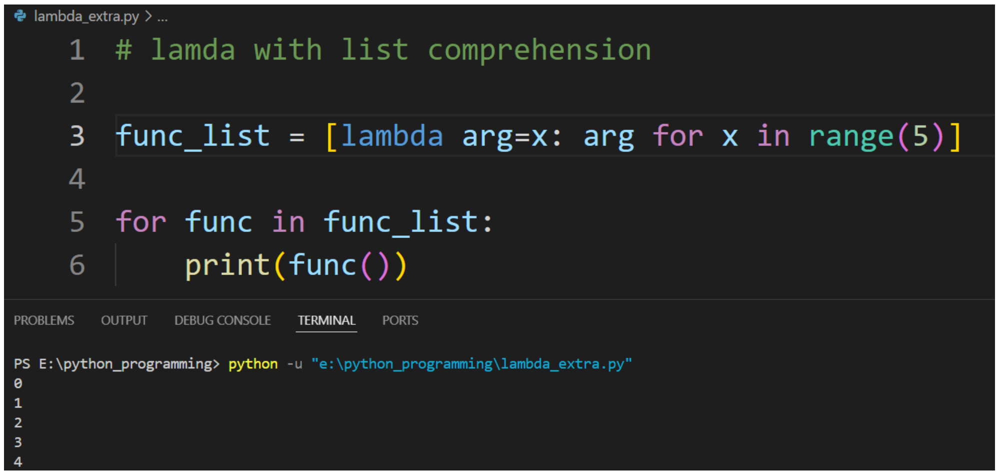
```python
numbers = list(map(int, input("Enter numbers seperated by space").split()))

is_even = lambda x: "Even" if x % 2 == 0 else "Odd"

for num in numbers:
    print(f"{num} is {is_even(num)}")
```

```
PS E:\python_programming> python -u "e:\python_programming\lamda_even_or_odd.py"
Enter numbers seperated by space1 22 29 1 29 30 45 76 89
1 is Odd
22 is Even
29 is Odd
1 is Odd
29 is Odd
30 is Even
45 is Odd
76 is Even
89 is Odd
```

The provided code is a compact and efficient way to read a list of integers from the user, determine whether each integer is even or odd using a lambda function, and print the result. The code begins by prompting the user to enter numbers separated by spaces. It reads the entire input line as a string, splits the string into a list of substrings (each representing a number), and then converts each substring to an integer using map(int, ...), which is then converted to a list. This results in a list of integers called numbers. The lambda function is_even takes an integer x as input and returns the string "Even" if x is divisible by 2 (using the condition x % 2 == 0), and "Odd" otherwise. A for loop iterates over each number in the numbers list, applies the is_even lambda function to determine if the number is even or odd, and prints the result in the format {num} is {is_even(num)}, where num is the current number from the list.


lamda_and_filter.py > ...

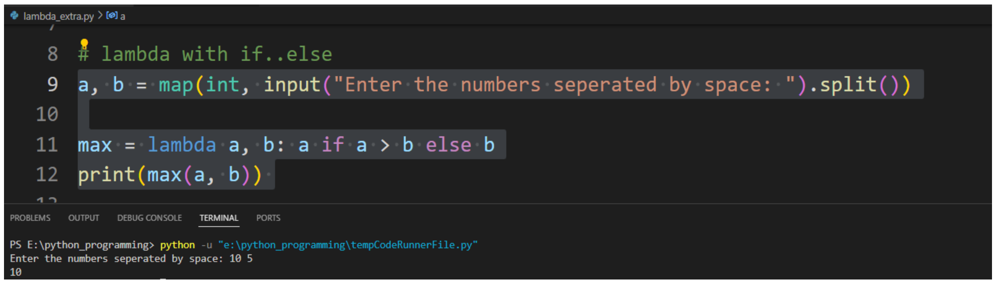
```python
# The filter() function in Python takes in a function and an
# iterable (lists, tuples, and strings) as arguments.
# The function is called with all the items in the list,
# and a new list is returned,
# which contains items for which the function evaluates to True.
numbers = [1, 10, 15, 12, 3, 16, 20, 5, 7]

even = list(filter(lambda x : (x % 2 == 0), numbers))
print(even)

double = list(map(lambda x: x * 2, numbers))
print(double)
```


lambda_extra.py > ...

```python
# lamda with list comprehension

func_list = [lambda arg=x: arg for x in range(5)]

for func in func_list:
    print(func())
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS E:\python_programming> python -u "e:\python_programming\lambda_extra.py"
0
1
2
3
4
```


lambda_extra.py > a

```python
# lambda with if..else
a, b = map(int, input("Enter the numbers seperated by space: ").split())

max = lambda a, b: a if a > b else b
print(max(a, b))
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS E:\python_programming> python -u "e:\python_programming\tempCodeRunnerFile.py"
Enter the numbers seperated by space: 10 5
10
```

8. Operations

Function	Description
len(L)	Returns the number of items in list L
max(L)	Returns the maximum value in list L
min(L)	Returns the minimum value in list L
sum(L)	Returns the sum of the values in list L
sorted(L)	Returns a copy of list L where the items are in order from smallest to largest (This does not mutate L.)

Method	Description
L.append(v)	Appends value v to list L.
L.clear()	Removes all items from list L.
L.count(v)	Returns the number of occurrences of v in list L.
L.extend(v)	Appends the items in v to L.
L.index(v)	Returns the index of the first occurrence of v in L—an error is raised if v doesn't occur in L.
L.index(v, beg)	Returns the index of the first occurrence of v at or after index beg in L—an error is raised if v doesn't occur in that part of L.
L.index(v, beg, end)	Returns the index of the first occurrence of v between indices beg (inclusive) and end (exclusive) in L; an error is raised if v doesn't occur in that part of L.
L.insert(i, v)	Inserts value v at index i in list L, shifting subsequent items to make room.
L.pop()	Removes and returns the last item of L (which must be nonempty).
L.remove(v)	Removes the first occurrence of value v from list L.
L.reverse()	Reverses the order of the values in list L.
L.sort()	Sorts the values in list L in ascending order (for strings with the same letter case, it sorts in alphabetical order).
L.sort(reverse=True)	Sorts the values in list L in descending order (for strings with the same letter case, it sorts in reverse alphabetical order).

Method	Description
S.add(v)	Adds item v to a set S—this has no effect if v is already in S
S.clear()	Removes all items from set S
S.difference(other)	Returns a set with items that occur in set S but not in set other
S.intersection(other)	Returns a set with items that occur both in sets S and other
S.issubset(other)	Returns True if and only if all of set S's items are also in set other
S.issuperset(other)	Returns True if and only if set S contains all of set other's items
S.remove(v)	Removes item v from set S
S.symmetric_difference(other)	Returns a set with items that are in exactly one of sets S and other—any items that are in both sets are *not* included in the result
S.union(other)	Returns a set with items that are either in set S or other (or in both)