Type casting and Data structures

1 Comments

Comments are used to explain what the code does, which is particularly useful for someone reading the code for the first time or for the original author when returning to the code after some time.

Types of comments:

- **Single-Line Comments**: Single-line comments start with the # symbol and extend to the end of the line. They are typically used for brief explanations.
- **Multi-Line Comments**: Multi-line comments can be written using triple quotes (''' or " " "). These are useful for longer explanations or f r commenting out blocks of code during debugging.

[15]:
```
# This is a single-line comment
x = 5  # This comment explains that x is assigned the value 5
```

[13]:
```
'''
This is a multi-line comment.
It can span multiple lines.
It is often used for longer explanations.
'''
y = 10
```

2 Type casting

Type casting is the process of converting a value from one data type to another. In Python, type casting is typically done using built-in functions that convert values to specific types. This is useful when you need to perform operations on values that require them to be of the same data type.

2.1 Other types to int

To type cast other types to an integer in Python, we use the built-in "int()" function.

[22]:
```
int(2.3)  # Float to int, we are giving 2.3 to the int function. It will return␣
 ↪2.
```

[22]:
```
2
```

```
[24]: int(2.3, 3.2)  # This will give an error, as we can't give two arguments to the␣
       ↪int function.
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[24], line 1
----> 1 int(2.3, 3.2)  # This will give an error, as we can't give two argument␣
 ↪to the int function.

TypeError: 'float' object cannot be interpreted as an integer
```

```
[26]: int(False)  # Boolean to int, we are giving False to the int function. It will␣
       ↪return 0, because it is internally referred to as 0.
```

```
[26]: 0
```

```
[28]: int(True)  # Boolean to int, we are giving True to the int function. It will␣
       ↪return 1, because it is internally referred to as 1.
```

```
[28]: 1
```

```
[9]: int('1')  # String to int, we are giving '1' to the int function. It will␣
      ↪return 1.
```

```
[9]: 1
```

```
[30]: int('1.5')  # This will give an error, as '1.5' is not a valid integer.
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[30], line 1
----> 1 int('1.5')  # This will give an error, as '1.5' is not a valid integer.

ValueError: invalid literal for int() with base 10: '1.5'
```

```
[32]: int('abc')  # This will give an error, as 'abc' is not a valid integer.
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[32], line 1
----> 1 int('abc')  # This will give an error, as 'abc' is not a valid integer.

ValueError: invalid literal for int() with base 10: 'abc'
```

```
[34]: int(1 + 2j)  # This will give an error, as complex numbers cannot be converted␣
      ↪directly to integers.
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[34], line 1
----> 1 int(1 + 2j)  # This will give an error, as complex numbers cannot be␣
 ↪converted directly to integers.

TypeError: int() argument must be a string, a bytes-like object or a real␣
 ↪number, not 'complex'
```

2.2 Other types to float

To type cast other types to an float in Python, we use the built-in "float()" function.

```
[39]: float(22)  # Integer to float, we are giving 22 to the float function. It will␣
      ↪return 22.0.
```

```
[39]: 22.0
```

```
[41]: float(True)   # Boolean to float. True is converted to 1.0.
```

```
[41]: 1.0
```

```
[43]: float(False)  # Boolean to float. False is converted to 0.0.
```

```
[43]: 0.0
```

```
[45]: float('2')  # String to float. The string '2' is converted to 2.0.
```

```
[45]: 2.0
```

```
[47]: float(12 + 3j)  # This will give an error, as complex numbers cannot be␣
      ↪converted directly to floats.
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[47], line 1
----> 1 float(12 + 3j)  # This will give an error, as complex numbers cannot be
 ↪converted directly to floats.

TypeError: float() argument must be a string or a real number, not 'complex'
```

2.3 Other types to complex

The complex() function is used to create complex numbers in Python. It can take either one or two arguments:

```
[32]: complex(1)  # int to complex. Creates a complex number with real part 1 and␣
       ↪imaginary part 0. Result: (1+0j)
```

```
[32]: (1+0j)
```

```
[36]: complex(2.0)  #float to complex. Creates a complex number with real part 2.0␣
       ↪and imaginary part 0. Result: (2.0+0j)
```

```
[36]: (2+0j)
```

```
[41]: complex(2.3, 3.2)   #float to complex with arguments. Creates a complex number␣
       ↪with real part 2.3 and imaginary part 3.2. Result: (2.3+3.2j)
```

```
[41]: (2.3+3.2j)
```

```
[54]: complex('2')  #string to complex. Creates a complex number with real part 2 and␣
       ↪imaginary part 0. Result: (2+0j)
```

```
[54]: (2+0j)
```

```
[58]: complex('2', '3')  # This will give an error because complex() can't take a␣
       ↪second argument if the first argument is a string.
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[58], line 1
----> 1 complex('2', '3')  # This will give an error because complex() can't␣
 ↪take a second argument if the first argument is a string.

TypeError: complex() can't take second arg if first is a string
```

```
[49]: complex(True)  # Creates a complex number with real part 1 and imaginary part 0.
       ↪ Result: (1+0j)
```

```
[49]: (1+0j)
```

```
[51]: complex(True, True)  # Creates a complex number with real part 1 and imaginary␣
       ↪part 1. Result: (1+1j)
```

```
[51]: (1+1j)
```

```
[61]: complex(False, False)  # Creates a complex number with real part 0 and␣
       ↪imaginary part 0. Result: (0+0j)
```

```
[61]: 0j
```

2.4 Other types to string

We can use the "str()" function to convert different data types to strings.

```
[58]: str(1)  # Converts the integer 1 to the string '1'.
```

```
[58]: '1'
```

```
[66]: str(3.14)  # Converts the float 3.14 to the string '3.14'.
```

```
[66]: '3.14'
```

```
[66]: str(1 + 22j)  # Converts the complex number 1 + 22j to the string '1+22j'.
```

```
[66]: '(1+22j)'
```

```
[68]: str(True)   # Converts True to the string 'True'.
```

```
[68]: 'True'
```

```
[70]: str(False)  # Converts False to the string 'False'.
```

```
[70]: 'False'
```

2.5 Other types bool

The "bool()" function in Python is used to convert various data types to boolean values

```
[74]: bool(0)   # Converts int 0 to False.
```

```
[74]: False
```

```
[72]: bool(1)   # Converts int 1 to True.
```

```
[72]: True
```

```
[76]: bool(0.0)    # Converts float 0.0 to False.
```

```
[76]: False
```

```
[78]: bool(3.14)   # Converts float 3.14 to True.
```

```
[78]: True
```

```
[80]: bool('')      # Converts an empty string to False.
```

```
[80]: False
```

```
[82]: bool('Hello') # Converts a non-empty string to True
```

```
[82]: True
```

```
[84]: bool(1 + 2j)  # Converts any non-zero complex number to True.
```

```
[84]: True
```

```
[86]: bool(0 + 0j)  # Converts zero complex number to False.
```

```
[86]: False
```

3 Data Structures

- **Data structures** are collections of data types.
- With **data types**, we can only assign one value to a variable.
- With **data structures**, we can assign multiple values to a variable.

3.1 Types of Data Structures

- **Built-in Data Structures**: List, Tuple, Set, Dictionary.
- **User-defind Data Structures**: Stack, Queue, LinkedList, Tree

3.2 List

1. **Ordered**: The elements in a list have a defined order, and this order will not change unless explicitly changed by some operation.
2. **Mutable**: The elements of a list can be changed (modified, added, removed) after the list is created.
3. **Heterogeneous**: Lists can contain elements of different data types (e.g., integers, strings, floats, other lists, etc.).
4. **Dynamic Size**: Lists can grow or shrink in size as needed.
5. **Indexable**: Elements in a list can be accessed using indices, with the first element having an index of 0.
6. **Slicing**: Parts of a list can be accessed or modified using slicing.
7. **Iterable**: Lists can be iterated over using loops.
8. **Duplicates Allowed**: Lists can contain duplicate elements.
9. **Methods**: Lists come with a variety of built-in methods for common operations, such as:
    - `append()`: Add an element to the end of the list.
    - `copy()`: Return a shallow copy of the list.
    - `insert()`: Add an element at a specified position.
    - `pop()`: Remove and return the element at the specified position (or the last element if no position is specified).
    - `remove()`: Remove the first occurrence of a specified element.
    - `index()`: Return the index of the first occurrence of a specified element.
    - `clear()`: Remove all elements from the list.
    - `extend()`: Add multiple elements to the end of the list.
    - `count()`: Return the number of occurrences of a specified element.

- `sort()`: Sort the elements of the list.
- `reverse()`: Reverse the order of elements in the list.

[609]:
```python
#List Creation
mylist  =[1,10,3,6,5,4]
diff_list = ['1','a',True, 1,1+2j, 1.4]# list can contain different data types
nested_list =[1,2,4,[1,34,45,1]]# list can another list and iterables
print(mylist)
print(diff_list)
print(nested_list)
```

```
[1, 10, 3, 6, 5, 4]
['1', 'a', True, 1, (1+2j), 1.4]
[1, 2, 4, [1, 34, 45, 1]]
```

3.2.1 Indexing

Indexing allows you to access individual elements in a list (or other iterable types like strings and tuples) using their position (index) within the list.

- Syntax: list[index]
- Forward Indexing: Python uses zero-based indexing, meaning the first element has an index of 0, the second element has an index of 1, and so on.
- Backward Indexing: Backward indexing counts from the end of the list. The last element has an index of -1, the second-to-last element has an index of -2, and so on.

[612]:
```python
mylist
```

[612]:
```
[1, 10, 3, 6, 5, 4]
```

[613]:
```python
#Example of indexing
# Access elements using positive indexing
print(mylist[0])  # Output: '1'
print(mylist[2])  # Output: '3'

# Access elements using negative indexing
print(mylist[-1])  # Output: '4'
print(mylist[-3])  # Output: '6'
```

```
1
3
4
6
```

[614]:
```python
#we can also modify values at specific indices by assigning a new value to the␣
 ↪indexed position.
mylist[0] =9
print(mylist)
```

```
[9, 10, 3, 6, 5, 4]
```

3.2.2 Slicing

Slicing allows you to access a subset of a list by specifying a start, stop, and step value.

- Syntax: list[start:stop:step]
- Start: The index where the slice starts (inclusive).
- Stop: The index where the slice ends (exclusive).
- Step: The interval between each index in the slice (optional, defaults to 1).

[617]:
```
#Example of slicing
# Define a list
l2 = ['a', 'b', 'c', 'd', 'e']

# Slice with start and stop
print(l2[1:4])  # Output: ['b', 'c', 'd']

# Slice with start, stop, and step
print(l2[0:5:2])  # Output: ['a', 'c', 'e']

# Slice with default step
print(l2[::2])  # Output: ['a', 'c', 'e']

# Slice with negative step (reverse)
print(l2[::-1])  # Output: ['e', 'd', 'c', 'b', 'a']

# Omitting start (defaults to 0)
print(l2[:3])  # Output: ['a', 'b', 'c']

# Omitting stop (defaults to the end of the list)
print(l2[2:])  # Output: ['c', 'd', 'e']
```

```
['b', 'c', 'd']
['a', 'c', 'e']
['a', 'c', 'e']
['e', 'd', 'c', 'b', 'a']
['a', 'b', 'c']
['c', 'd', 'e']
```

3.2.3 append()

append() method in Python is used to add a single element to the end of a list at a time.

[622]:
```
mylist.append(100)
```

[624]:
```
mylist # we can see that value 100 is add at end of the list
```

```
[624]: [9, 10, 3, 6, 5, 4, 100]
```

```
[625]: mylist(4,5)# append() functione does not take two arguments
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[625], line 1
----> 1 mylist(4,5)# append() functione does not take two arguments

TypeError: 'list' object is not callable
```

3.2.4 copy()

Return a shallow copy of the list.

```
[629]: l1= mylist.copy()
```

```
[631]: l1 # we can see the mylist is copied to l1
```

```
[631]: [9, 10, 3, 6, 5, 4, 100]
```

3.2.5 clear()

- Purpose: Remove all elements from a list.
- Effect: The list becomes empty, with a length of zero, but the list object still exists.

```
[634]: l1.clear()
       print(l1)# we can see the data from l1 is cleared
```

```
[]
```

3.2.6 insert()

- Add an element at a specified position.
- Syntax: list_name.insert(index, element)
- The specified element is inserted before the given index.

```
[637]: mylist
```

```
[637]: [9, 10, 3, 6, 5, 4, 100]
```

```
[639]: mylist.insert(2,99)# we can see before given index 2 the value 99 is inserted
       mylist
```

```
[639]: [9, 10, 99, 3, 6, 5, 4, 100]
```

```
[640]: mylist.insert(-1,22) # we can see befire  given index -1 the value 22 is␣
        ↪inserted
```

```
mylist
```

```
[640]: [9, 10, 99, 3, 6, 5, 4, 22, 100]
```

3.2.7 pop()

- Purpose: Remove and return an element from a list.
- Syntax: list_name.pop([index]
- Effect: If no index is specified, the last element is removed and returned. If an index is specified, the element at that index is removed and returned.

```
[644]: mylist
```

```
[644]: [9, 10, 99, 3, 6, 5, 4, 22, 100]
```

```
[645]: mylist.pop() # by defualt it removes and return the last element
```

```
[645]: 100
```

```
[647]: mylist
```

```
[647]: [9, 10, 99, 3, 6, 5, 4, 22]
```

```
[648]: mylist.pop(7) # the element at index 7 is removed
```

```
[648]: 22
```

```
[650]: mylist
```

```
[650]: [9, 10, 99, 3, 6, 5, 4]
```

```
[651]: mylist.pop(10)# it will raise error beacuase index 10 is not there
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[651], line 1
----> 1 mylist.pop(10)# it will raise error beacuase index 10 is not there

IndexError: pop index out of range
```

3.2.8 remove()

- Purpose: Remove the first occurrence of a specified value from a list.
- Syntax: list_name.remove(value)
- Effect: The specified value is removed from the list. If the value appears multiple times, only the first occurrence is removed. If the value is n t found, a ValueError is raised

[655]:
```
mylist
```

[655]:
```
[9, 10, 99, 3, 6, 5, 4]
```

[657]:
```
mylist.remove(9)# the value 9 is removed
mylist
```

[657]:
```
[10, 99, 3, 6, 5, 4]
```

[659]:
```
mylist.remove(4)# the value 9 is removed
mylist
```

[659]:
```
[10, 99, 3, 6, 5]
```

[661]:
```
#let see the example for duplicate values how it will behave
mylist.insert(1,5) #   added 5 value before index 1, now we duplicte  value␣
 ↪whichi is 5
mylist
```

[661]:
```
[10, 5, 99, 3, 6, 5]
```

[663]:
```
mylist.remove(5) # we have two 5 in  list, it only remove the first occurence
mylist
```

[663]:
```
[10, 99, 3, 6, 5]
```

[665]:
```
mylist.remove(100) # the specified value is not there in remove it raise error
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[665], line 1
----> 1 mylist.remove(100) # the specified value is not there in remove it rais␣
 ↪error

ValueError: list.remove(x): x not in list
```

3.2.9 index()

- Purpose: Return the index of the first occurrence of a specified element.
- Syntax: list_name.index(value)
  - value: The value whose index you want to fin at.

[668]:
```
mylist
```

[668]:
```
[10, 99, 3, 6, 5]
```

```
[669]: mylist.index(99) # the index for value 99 is 1
```

```
[669]: 1
```

```
[671]: # let's check for duplicate value how the fuction will behave
       mylist.append(10)
```

```
[673]: mylist
```

```
[673]: [10, 99, 3, 6, 5, 10]
```

```
[674]: mylist.index(10) #index of first occurrence of a specified element
```

```
[674]: 0
```

```
[680]: mylist.index(100)# the element is not there so it will raise error
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[680], line 1
----> 1 mylist.index(100)# the element is not there so it will raise error

ValueError: 100 is not in list
```

3.2.10 extend()

- Purpose: Add all elements from another iterable (like a list or a string) to the end of the current list.
- Syntax: list_name.extend(iterable)

```
[683]: l1=[1,2,3,4]
       l2=[5,6,7,8]
```

```
[685]: l1.extend(l2)# it add element of l2 to l1
       l1
```

```
[685]: [1, 2, 3, 4, 5, 6, 7, 8]
```

3.2.11 count()

- Purpose: Count how many times a specified value appears in a list.
- Syntax: list_name.count(value)
- value: The value whose occurrences you want to count.

```
[688]: lis = [1,2,3,1,3,4,5,5]
```

```
[690]: lis
```

```
[690]: [1, 2, 3, 1, 3, 4, 5, 5]
```

```
[692]: lis.count(2)# value 2 appeared in list one time
```

```
[692]: 1
```

```
[693]: lis.count(1) # value 1 appeared in list two times
```

```
[693]: 2
```

```
[695]: lis.count(10) # value 10 is not there in list so i will give zero
```

```
[695]: 0
```

3.2.12 Sort()

- Purpose: Sort the elements of a list in place.- Syntax: list_name.sort(key=None, reverse=False
- key (optional): A function that takes one argument and returns a value to be used for sorting purpos
- `reverse (optional): A boolean. If True, the list is sorted in descending or`

  er. If False (the default), the list is sorted in ascending order.

```
[699]: mylist
```

```
[699]: [10, 99, 3, 6, 5, 10]
```

```
[700]: mylist.sort() # by defualt it sort element ascending order
       mylist
```

```
[700]: [3, 5, 6, 10, 10, 99]
```

```
[701]: mylist.sort(reverse =True)# if we change the reverse value= True, it will␣
         ↪perform descending.
       mylist
```

```
[701]: [99, 10, 10, 6, 5, 3]
```

3.2.13 reverse()

- Purpose: Reverse the order of elements in a list in place.
- Syntax: list_name.reverse()

```
[706]: lis
```

```
[706]: [1, 2, 3, 1, 3, 4, 5, 5]
```

```
[708]: lis.reverse()
       lis
```

```
[708]: [5, 5, 4, 3, 1, 3, 2, 1]
```

```
[710]: lis.reverse()
       lis
```

```
[710]: [1, 2, 3, 1, 3, 4, 5, 5]
```

3.2.14 all()

The all function in Python returns True if all elements in the given iterable (e.g., list, tuple, set) are True. If the iterable is empty, all also returns True. all will return False if at least one element in the iterable is False.

```
[713]: l1
```

```
[713]: [1, 2, 3, 4, 5, 6, 7, 8]
```

```
[715]: all(l1)# All numbers are non-zero, which means they are `True`
```

```
[715]: True
```

```
[716]: l3=[1,2,3,0]
       all(l3)# one element is zero, which means they are `False`
```

```
[716]: False
```

3.2.15 any()

- The any function in Python returns True if at least one element in the given iterable is True. If the iterable is empty, any returns False.

```
[720]: l1
```

```
[720]: [1, 2, 3, 4, 5, 6, 7, 8]
```

```
[721]: any(l1)
```

```
[721]: True
```

```
[722]: l4=[]
       any(l4)
```

```
[722]: False
```

3.2.16 list membership

List membership refers to checking whether a specific value exists within a list. In Python, you can use the **"in"** keyword to test if an item is present in a list.

```
[724]: l1
```

```
[724]: [1, 2, 3, 4, 5, 6, 7, 8]
```

```
[725]: 1 in l1 # checks the 1 is present  in l1 or not
```

```
[725]: True
```

```
[731]: 8 in l1 # checks the 8 is present  in l1 or not
```

```
[731]: True
```

```
[733]: 10 in l1 # checks the 10 is present  in l1 or not
```

```
[733]: False
```

3.2.17 enumerate()

- Purpose: To get both the index (position) and the value of each item in a list at the same time

```
[736]: mylist
```

```
[736]: [99, 10, 10, 6, 5, 3]
```

```
[738]: for i in enumerate(mylist):
           print(i)
```

```
(0, 99)
(1, 10)
(2, 10)
(3, 6)
(4, 5)
(5, 3)
```

```
[ ]:                                                # To be continued...
```