Built-in Data Types in Python

(Number,List,Tuple,String)

Massive Information & Knowledge Engineering

204113 Computer & Programming

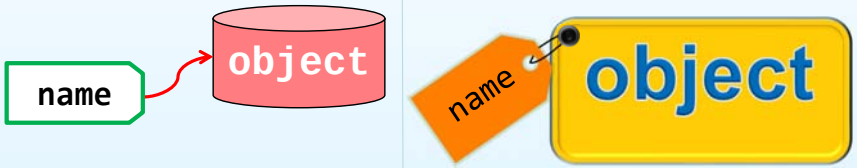
Dr. Arnon Rungsawang
Dept. of computer engineering
Kasetsart University

https://mike.cpe.ku.ac.th/204113

Version 2025

What Is a Variable?

- Any values such as 85, 23+8, "Hello" are objects that get stored inside computer's memory
- A variable is like a name tag given to such an object.

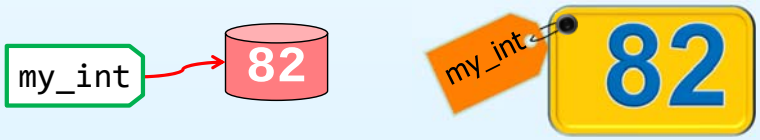


Binding Variables to Values

- An assignment statement (=) creates a new variable, gives it a value (an object in memory), and binds a name to the value

```
my_int = 82
```

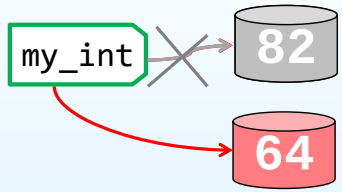
  - the variable name (my_int)
  - the assignment operator, also known as the equal sign (=)
  - the object that is being tied to the variable name (82)



Changing Bindings

- Binding can be changed with a new assignment statement
- Previous objects may still reside in memory, but will be removed later on

```
my_int = 82
my_int = 64
```



Python Data Types

- In computer programming, data types specify the type of data that can be stored inside a variable.

```
num = 24
```

  - Here, `24` (an integer) is assigned to the `num` variable. So the data type of `num` is of the `int` class.

Data Types	Classes	Description
Numeric	`int, float, complex`	Holds numeric values
String	`str`	Holds sequence of characters
Sequence	`list, tuple, range`	Holds collection of items
Mapping	`dict`	Holds data in key-value pair form
Boolean	`bool`	Holds either `True` or `False`
Set	`set, fronzenset`	Holds collection of unique items

Numeric Data Types

- Numeric data type is used to hold numeric values.
  - `int` – holds signed integers of non-limited length
  - `float` – holds floating decimal points and it's accurate up to 15 decimal places
  - `complex` – holds complex numbers
- We can use the `type()` function to know which class a variable or a value belongs to.

```
1  num1 = 5
2  print(num1, 'is of type', type(num1))
3
4  num2 = 2.0
5  print(num2, 'is of type', type(num2))
6
7  num3 = 1+2j
8  print(num3, 'is of type', type(num3))
```

Number systems

- Sometime, computer programmers need to work with binary **(base 2)**, hexadecimal **(base 16)** and octal **(base 8)** number systems.
- In Python, we can represent these numbers by appropriately placing a prefix before that number. The following table lists these prefixes.

Number System	Prefix
Binary	`0b` or `0B`
Octal	`0o` or `0O`
Hexadecimal	`0x` or `0X`

```
1  print(0b1101011)  # prints 107
2
3  print(0xFB + 0b10)  # prints 253
4
5  print(0o15)  # prints 13
```

Type conversion

- In programming, type conversion is the process of converting one type of number into another.
- Operations like addition, subtraction convert integers to float implicitly (automatically), if one of the operands is float.

```
1  print(1 + 2.0) # prints 3.0
```

Explicit type conversion

- We can also use built-in function like `int()`, `float()` and `complex()` to convert between types explicitly.
- These functions can even convert from `string`s.

```
1  num1 = int(2.3)
2  print(num1)  # prints 2
3
4  num2 = int(-2.8)
5  print(num2)  # prints -2
6
7  num3 = float(5)
8  print(num3) # prints 5.0
9
10 num4 = complex('3+5j')
11 print(num4)  # prints (3 + 5j)
```

- Here, when converting from float to integer, the number gets truncated (decimal parts are removed.
- Similarly, when converting from integer to float, `.0` is postfixed to the number.

Python Lists

Python Lists

- In Python, lists are used to store multiple data at once.
- Suppose we need to record the scores of 5 students. Instead of creating 5 separate variables, we can simply create a list:

75	80	63	42	91

```
# a list with 5 integers
scores = [75, 80, 63, 42, 91]
print(scores)

for i in range(len(scores)):
    print(scores[i], end=' ')
print()
```


Members in a list

- A list can have any number of items and they may be of different types (integer, float, string, bool, etc.).

```
my_list = [1, "Hello", 3.4, True, None]

# empty list
my_list = []

# list with duplicate elements
my_list = [1, "Hello", 3.4, "Hello", 1]

# list with list or tuple
scores = [["Somchai", ("Physic I", 75), ("Math I", 67)],
          ["Somkid", ("Math I", 84), ("Thai Com", 93)]]

# How to access elements in the above list?
```

Negative indexing in Python

- Python allows negative indexing for its sequences.
  - The index of **-1** refers to the last item, **-2** to the second last item and so on.
- Note that if the specified index does not exist in a list, Python throws the `IndexError` exception.

```
languages = ["Python", "Swift", "C++"]

# access item at index 0
print(languages[-1])   # C++

# access item at index 2
print(languages[-3])   # Python
```

	"Python"	"Swift"	"C++"
index	0	1	2
negative index	-3	-2	-1

Slicing of a list

```
# List slicing in Python

my_list = ['p','r','o','g','r','a','m','i','z']

# items from index 2 to index 4
print(my_list[2:5])

# items from index 5 to end
print(my_list[5:])

# items beginning to end
print(my_list[:])
```

- In Python, it is possible to access a portion of a list using the slicing operator `:`.
- In the left example,
  - `mylist[2:5]` returns a list with items from index 2 to index 4.
  - `mylist[5:]` returns a list with items from index 5 to the end.
  - `mylist[:]` returns all list items.
- Note that when we slice lists, the start index is inclusive, but the end index is exclusive.

Add elements to a list

```
numbers = [21, 34, 54, 12]

print("Before Append:", numbers)

# using append method
numbers.append(32)

print("After Append:", numbers)
```

```
numbers = [1, 3, 5]

even_numbers = [4, 6, 8]

# add elements of even_numbers to the
    numbers list
numbers.extend(even_numbers)

print("List after append:", numbers)
```

```
numbers = [10, 30, 40]

# insert an element at index 1 (second
    position)
numbers.insert(1, 20)

print(numbers) # [10, 20, 30, 40]
```

- Lists are mutable (changeable) object.
  - Meaning we can add and remove elements from a list.
- Python list provides different methods to add items to a list.
  - The `append()` method adds an item at the end of the list.
  - We use the `extend()` method to add all the items of an iterable (list, tuple, string, dictionary, etc.) to the end of the list.
  - We use the `insert()` method to add an element at the specified index.

Change list items

- Python lists are mutable. Meaning lists are changeable.
- We can change items of a list by assigning new values using the `=` operators.

```
languages = ['Python', 'Swift', 'C++']

# changing the third item to 'C'
languages[2] = 'C'

print(languages)  # ['Python', 'Swift', 'C']
```

Remove an item from a list

```
languages = ['Python', 'Swift', 'C++', 'C',
    'Java', 'Rust', 'R']

# deleting the second item
del languages[1]
print(languages) # ['Python', 'C++', 'C',
    'Java', 'Rust', 'R']

# deleting the last item
del languages[-1]
print(languages) # ['Python', 'C++', 'C',
    'Java', 'Rust']

# delete the first two items
del languages[0 : 2]  # ['C', 'Java',
    'Rust']
print(languages)
```

- In Python we can use the `del` statement to remove one or more items from a list.

```
languages = ['Python', 'Swift', 'C++', 'C',
    'Java', 'Rust', 'R']

# remove 'Python' from the list
languages.remove('Python')

print(languages) # ['Swift', 'C++', 'C',
    'Java', 'Rust', 'R']
```

- We can also use the `remove()` method to delete a list item.

List methods

Method	Description
`append()`	add an item to the end of the list
`extend()`	add all the items of an iterable to the end of the list
`insert()`	inserts an item at the specified index
`remove()`	removes item present at the given index
`pop()`	returns and removes item present at the given index
`clear()`	removes all items from the list
`index()`	returns the index of the first matched item
`count()`	returns the count of the specified item in the list
`sort()`	sort the list in ascending/descending order
`reverse()`	reverses the item of the list
`copy()`	returns the shallow copy of the list

Iterating through a list

- We can use a `for` loop to iterate over the elements of a list.

```
languages = ['Python', 'Swift', 'C++']

# iterating through the list
for language in languages:
    print(language)
```


Check if an element exists in a list

- We use the `in` keyword to check if an item exists in the list or not.

```
languages = ['Python', 'Swift', 'C++']

print('C' in languages)    # False
print('Python' in languages)    # True
```

List length and List comprehension

- We use the `len` function to find the size of a list.

```
languages = ['Python', 'Swift', 'C++']

print("List: ", languages)

print("Total Elements: ", len(languages))     # 3
```

- List comprehension is a concise and elegant way to create lists.

```
# create a list with value n ** 2
# where n is a number from 1 to 5
numbers = [n**2 for n in range(1, 6)]

print(numbers)

# Output: [1, 4, 9, 16, 25]
```


List Comprehension

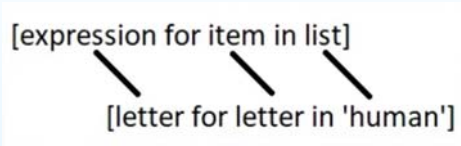
List comprehension vs. `for` loop

- Suppose we want to separate the letters of the word `'human'` and add the letters as items of a list. The first thing comes in mind would be using a `for` loop.
- However, Python has an easier way to solve this issue using list comprehension.
- List comprehension is an elegant way to define and create lists based on existing lists.

```
h_letter1 = []

for letter in 'human':
    h_letter1.append(letter)

print(h_letter1)

# using list comprehension
h_letter2 = [ letter for letter in 'human' ]
print( h_letter2)
```


List comprehension syntax

- If we noticed, `'human'` is a string, not a list. This is the power of list comprehension. It can identify when it receives a string or a tuple and iterates on it like a list.




```
# using list comprehension
h_letter2 = [ letter for letter in 'human' ]
```

List comprehension vs. Lambda function

- List comprehensions aren't the only way to work on lists.
- Various built-in functions and lambda functions can create and modify lists in less lines of code.
  - Here, we use lambda function and string as arguments of a `map()` function.

```
1  m = map(lambda x: x+1, range(10))
2  print(type(m))
3  print(list(m), '\n')
4
5  letters = list(map(lambda x: x, 'human'))
6  print(letters)
7
8  p = [x for x in 'human']
9  print(p)
```

  - However, list comprehensions are usually more human readable than lambda functions. It is easier to understand what the programmer was trying to accomplish when list comprehensions are used.

Conditional in list comprehension

- List comprehensions can utilize conditional statement to modify existing list (or other tuples). We will create list that uses mathematical operators, integers, and `range()`.

```
1  number_list = [ x for x in range(20) if x % 2 == 0]
2  print(number_list)
3
4  num_list = [y for y in range(100) if y % 2 == 0 and y % 5 == 0]
5  print(num_list)
6
7  obj = ["Even" if i%2==0 else "Odd" for i in range(10)]
8  print(obj)
```

Nested loop in list comprehension

- Suppose, we need to compute the transpose of a matrix that required nested `for` loop.

```
1  transposed = []
2  matrix = [[1, 2, 3, 4], [5, 6, 7, 8]]
3  for i in range(len(matrix[0])):
4      transposed_row = []
5      for row in matrix:
6          transposed_row.append(row[i])
7      transposed.append(transposed_row)
8  print(transposed)
9  """------------------------"""
10 # matrix = [[1, 2, 3, 4], [5, 6, 7, 8]]
11 # transpose = [[row[i] for row in matrix] for i in range(4)]
12 # print (transpose)
```

List comprehension – key points

- List comprehension is an elegant way to define and create lists based on existing lists.
- List comprehension is generally more compact and faster than normal functions and loops for creating list.
- However, we should avoid writing very long list comprehensions in one line to ensure that code is user-friendly.
- Remember, every list comprehension can be rewritten in `for` loop, but every `for` loop can't be rewritten in the form of list comprehension.

Python Tuples

Python Tuples

- A tuple in Python is similar to a list.
- The difference between the two is that we cannot change the elements of a tuple once it is assigned (i.e., immutable) whereas we can change the elements of a list.

Creating a tuple

```
1  # Different types of tuples
2
3  # Empty tuple
4  my_tuple = ()
5  print(my_tuple)
6
7  # Tuple having integers
8  my_tuple = (1, 2, 3)
9  print(my_tuple)
10
11 # tuple with mixed datatypes
12 my_tuple = (1, "Hello", 3.4)
13 print(my_tuple)
14
15 # nested tuple
16 my_tuple = ("mouse", [8, 4, 6], (1, 2, 3))
17 print(my_tuple)
18
19 my_tuple2 = 'un', 'deux', 'trois'
20 my_tuple3 = 1, "Hello", 3.1416
```

- A tuple is created by placing all the items (elements) inside parentheses ( ), separated by commas. The parentheses are optional; however, it is a good practice to use them.
- A tuple can have any number of items, and they may be of different types (integer, float, list, string, etc.).
- Note that the last two statements illustrate how we can also create tuples without using parentheses.

Create a tuple with one element

- In Python, creating a tuple with one element is a bit tricky. Having one element within parentheses is **not** enough.
- We will need a trailing comma to indicate that it is a tuple.

```
1  var1 = ("hello")
2  print(type(var1))  # <class 'str'>
3
4  # Creating a tuple having one element
5  var2 = ("hello",)
6  print(type(var2))  # <class 'tuple'>
7
8  # Parentheses is optional
9  var3 = "hello",
10 print(type(var3))  # <class 'tuple'>
11
12 # whether var2 is equivalent to var3
13 print(var2==var3)
```

Access tuple elements

- Like a list, each element of a tuple is represented by index numbers **(0, 1, ...)** where the first element is at index **0**.
- We use the index number to access tuple elements.

```
# accessing tuple elements using indexing
letters = ("p", "r", "o", "g", "r", "a", "m", "i", "z")
print(letters[0])   # prints "p"
print(letters[5])   # prints "a"

# accessing tuple elements using negative indexing
letters = ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')
print(letters[-1])   # prints 'z'
print(letters[-3])   # prints 'm'
```

Slicing a tuple

- Like list, we can access a range of items in a tuple by using the slicing operator colon :.

```
# accessing tuple elements using slicing
my_tuple = ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')

# elements 2nd to 4th index
print(my_tuple[1:4])  #  prints ('r', 'o', 'g')

# elements beginning to 2nd
print(my_tuple[:-7]) # prints ('p', 'r')

# elements 8th to end
print(my_tuple[7:]) # prints ('i', 'z')

# elements beginning to end
print(my_tuple[:]) # Prints ?
```

Tuple methods

- In Python ,methods that add items or remove items are not available with tuple.
- Only the following two methods are available.

```
my_tuple = ('a', 'p', 'p', 'l', 'e',)

print(my_tuple.count('p'))  # prints 2
print(my_tuple.index('l'))  # prints 3
```

Iterate and check if there exist an element

- We can use the for loop to iterate over the elements of a tuple.

```
languages = ('Python', 'Swift', 'C++')

# iterating through the tuple
for language in languages:
    print(language)
```

- We use the in keyword to check if an item exists in the tuple or not.

```
languages = ('Python', 'Swift', 'C++')

print('C' in languages)    # False
print('Python' in languages)    # True
```

Advantages of tuple over list in Python

- Since tuples are quite similar to lists, both of them are used in similar situations.
- However, there are certain advantages of implementing a tuple over a list:
  - We generally use tuples for heterogeneous (different) data types and lists for homogeneous (similar) data types.
  - Since tuples are immutable, iterating through a tuple is faster than with a list. So, there is a slight performance boost.
  - Tuples that contain immutable elements can be used as a key for a dictionary. With lists, this is not possible.
  - If you have data that doesn't change, implementing it as tuple will guarantee that it remains write-protected.

Python String

Strings

- In computer programming, a string is a sequence of characters.
- In Python, there is **no** notion of character. Therefore, a Python string is a sequence of consecutive one character string.
- We use single or double quotes to delimit a string in Python.

```
# create a string using single quote
string1 = 'I love "Python".'

# create a string using double quote
string2 = "Python's much fun."

print(string1, string2)
```

Negative index and slicing

- Similar to a list, Python allow negative indexing for its strings.

```
1 s = 'I love Python'
2 for i in range(-len(s), 0):
3     c = s[i]
4     print(c, end='')
5 print()
```

- Access a range of character strings in a string can also be done by slicing operator colon :.

```
1 s = 'I love Python'
2 ss = s[s.index('P'):]
3 print(ss)
```

Strings are immutable

- In python, strings are immutable. That means the characters of a string once created cannot be changed.
- However, we can assign the variable name to a new string.

```
message = 'Hola Amigos'
# TypeError: 'str' object does not support
# item assignment
# message[0] = 'H'
print(message)

# assign new string to message variable
message = 'Hello Friends'

print(message); # prints "Hello Friends"
```

Multiline string

- We can create a multiline string using triple double or single quotes.

```
message = """
  When the night has come
  And the land is dark
  And the moon is the only light we'll see
  No, I won't be afraid
  Oh, I won't be afraid
  Just as long as you stand
  Stand by me

  So darlin', darlin', stand by me
  Oh, stand by me
  Oh, stand
  Stand by me, stand by me
"""
print(message)
```

String operations

- There are many operations that can be performed with strings which makes it one of the most used data types in Python.
- Compare two strings
  - We use the == operator to compare two strings. If two strings are equal, the operator return `True`. Otherwise, it returns `False`.

```
str1 = "Hello, world!"
str2 = "I love Python."
str3 = "Hello, world!"

# compare str1 and str2
print(str1 == str2)

# compare str1 and str3
print(str1 == str3)
```

String operations (2)

- Join two or more strings
  - We can join (concatenate) two or more string using + operator.

```
greet = "Hello, "
name = "Jack"

# using + operator
result = greet + name
print(result)

# Output: Hello, Jack
```

Iterate through a string

- We can iterate through a string using a `for` loop.

```
1  s = 'Hello Python'
2  i = 0
3
4  for _ in s:
5      print(' '*i + s[:i+1])
6      i += 1
7  i = len(s)-2
8  for _ in s:
9      print(' '*i + s[:i+1])
10     i -= 1
```


String membership test

- We can test if a substring exists within a string or not, using the keyword `in`.

```
1  message = """
2    When the night has come
3    And the land is dark
4    And the moon is the only light we'll see
5    No, I won't be afraid
6    Oh, I won't be afraid
7    Just as long as you stand
8    Stand by me
9    So darlin', darlin', stand by me
10   Oh, stand by me
11   Oh, stand
12   Stand by me, stand by me
13 """
14 m = message.split('\n')
15 count = 0
16 for line in m:
17     if 'oh' in line.lower():
18         count += 1
19 print(f'Found \'oh\': {count}')
```


String methods

Methods	Description
`upper()`	Converts string to uppercase
`lower()`	Converts string to lowercase
`replace()`	Replaces substring inside
`find()`	Return index of the first string occurrence
`rstring()`	Removes trailing characters
`split()`	Splits string from left
`startswith()`	Checks if strings starts with a string
`isnumeric()`	Checks numeric characters
`index()`	Returns index of substring

Escapes sequences

Escape sequence	Description
`\\`	Backslash
`\'`	Single quote
`\"`	Double quote
`\a`	ASCII Bell
`\b`	ASCII Blackspace
`\f`	ASCII Formfeed
`\n`	ASCII Linefeed
`\r`	ASCII Carriage Return
`\t`	ASCII Horizontal Tab
`\v`	ASCII Vertical Tab
`\oYY`	Character with octal value YY
`\xHH`	Character with hexadecimal value HH

String Formatting (f-strings)

- Python f-Strings make it easy to print values and variables.

```
1  name = 'Cathy'
2  country = 'UK'
3
4  print(f'{name} is from {country}')
```

- Here, `f'{name} is from {country}'` is an f-string.

Problem Solving Samples

Task: Score statistics

- Read a list of scores and report the summary table, along with average, minimum, and maximum scores.


24


28


26


32

```
Enter student score (or ENTER to finish): 24
Enter student score (or ENTER to finish): 26
Enter student score (or ENTER to finish): 28
Enter student score (or ENTER to finish): 32
Enter student score (or ENTER to finish):
Student #1 score: 24.0
Student #2 score: 26.0
Student #3 score: 28.0
Student #4 score: 32.0
Average score is 27.5
Minimum score is 24.0
Maximum score is 32.0
```


Score statistics – Ideas

- We first build te two subroutines `read_scores()` and `compute_average()`.
- min and max can be computed using the built-in functions.
- The only challenge is the summary table part, `show_scores_summary(scores)`.

```
scores = read_scores()
show_scores_summary(scores)
avg_score = compute_average(scores)
min_score = min(scores)
max_score = max(scores)
print(f"Average score is {avg_score}")
print(f"Minimum score is {min_score}")
print(f"Maximum score is {max_score}")
```

Score statistics – Program part I

```python
def read_scores():
    scores = []
    while True:
        score = input("Enter student score (or ENTER to finish): ")
        if score == "":
            break
        scores.append(float(score))
    return scores

def compute_average(scores):
    return sum(scores)/len(scores)

## main begins here
scores = read_scores()
avg_score = compute_average(scores)
# print(scores, avg_score)
```

```
Enter student score (or ENTER to finish): 24
Enter student score (or ENTER to finish): 26
Enter student score (or ENTER to finish): 28
Enter student score (or ENTER to finish): 32
Enter student score (or ENTER to finish):
Student #1 score: 24.0
Student #2 score: 26.0
Student #3 score: 28.0
Student #4 score: 32.0
Average score is 27.5
Minimum score is 24.0
Maximum score is 32.0
```


Score statistics – Program part II

```python
def show_scores_summary(scores):
    for i in range(len(scores)):
        print(f"Student #{i+1} score: {scores[i]}")

show_scores_summary([31,56,73,48])
```

```
>>> show_scores_summary([31,56,73,48])
Student #1 score: 31
Student #2 score: 56
Student #3 score: 73
Student #4 score: 48
```


Max and Min value in a list

- Given a list `a = [12,37,5,19,3,7,15]`, find the max and min value.

```python
a = [12,37,5,19,3,7,15]

mymin = 9999999999
mymax = -mymin
len_a = len(a)

for i in range(len_a):
  if a[i] > mymax:
    mymax = a[i]
  if a[i] < mymin:
    mymin = a[i]

print(f'max: {mymax}, min: {mymin}')
print(max(a), min(a))
```


Simple numeric sorting in a list

- Give a list `a = [12,37,5,19,3,7,15]`, find the sorted descending list of `a`.

```python
def findMax(a):
  ''' return the index i where a[i] is max '''
  mymax = -9999999999
  len_a = len(a)
  res = -1
  for i in range(len_a):
    if a[i] > mymax:
      mymax = a[i]
      res = i
  return res

def swap(a, i, j):
  tmp = a[i]
  a[i] = a[j]
  a[j] = tmp

### main begins here
a = [12,37,5,19,3,7,15]

len_a = len(a)
for i in range(len_a):
  #print(a)
  b = a[i:]
  j = findMax(b)
  #print(f' i:{i}, b: {b}, j:{j}')
  swap(a, i, i+j)

print(a)
```

Tuple of a decimal digit

- Input an integer into a variable x and create a tuple that keeps each of its digits.

```
x = int(input('Input an integer: '))
y = []
while x > 0:
  y.append(x%10)
  x //=10
z = tuple(y[-1::-1]) # reverse the y's items
print(z)
```

Average value of numbers in a tuple

- Write a program to calculate the average value of the numbers in a give tuple of tuples.

```
nums = ((10, 10, 10, 12), (30, 45, 56, 45),
        (81, 80, 39, 32), (1, 2, 3, 4))

def myNums(nums):
    n = []
    for i in range(len(nums)):
        tmp = []
        for y in nums:
            tmp.append(y[i])
        n.append(tmp)
    return n

m1 = myNums(nums)
m2 = [[y[i] for y in nums] for i in range(len(nums))]
print(f'{m1}\n{m2}')

res = [sum(x)/len(x) for x in m1]
print(res)
```

```
>>> %Run test.py
  [[10, 30, 81, 1], [10, 45, 80, 2], [10, 56, 39, 3], [12, 45, 32, 4]]
  [[10, 30, 81, 1], [10, 45, 80, 2], [10, 56, 39, 3], [12, 45, 32, 4]]
  [30.5, 34.25, 27.0, 23.25]
```

String index

```
s = 'Python String'

ss = []
tmp = [x for x in s]
ss.append(tmp)
tmp = [i for i in range(len(s))]
ss.append(tmp)
tmp = [i for i in range(-len(s),1)]
ss.append(tmp)

for i in range(len(ss)):
    for j in range(len(s)):
        print(f'{ss[i][j]:>3}', end=' ')
    print()

print(f's[-13:-7]={s[-13:-7]}')
```

```
>>> %Run test.py
   P   y   t   h   o   n       S   t   r   i   n   g
   0   1   2   3   4   5   6   7   8   9  10  11  12
 -13 -12 -11 -10  -9  -8  -7  -6  -5  -4  -3  -2  -1
 s[-13:-7]=Python
```

Lowercase a string

```
s = 'Python String'

s1 = []
for c in s:
    if ord(c)>=ord('A') and ord(c)<=ord('Z'):
        s1.append(chr(ord('a')+ord(c)-ord('A')))
        continue
    s1.append(c)

s2 = ''
for c in s1:
    s2 += c
print(s2)
```

```
>>> %Run test.py
 python string
```

Count repeated characters in a string

- Write a program to count repeated characters in a string.

```
1  def countChar(str1):
2      s = []
3      sc = []
4      for c in str1:
5          if c in s:
6              sc[s.index(c)] += 1
7          else:
8              s.append(c)
9              sc.append(1)
10     return s, sc
11 ## main begins here
12 str1 = 'thequickbrownfoxjumpsoverthelazydog'
13 s, sc = countChar(str1)
14 # print(f'{s}\n{sc}')
15 for i in range(len(sc)):
16     if sc[i] > 1:
17         print(f'{s[i]}:{sc[i]}', end=' ')
18 print()
```

```
>>> %Run test.py
 t:2 h:2 e:3 u:2 r:2 o:4
```


To be continue..

つづく