# Lec 9. Introduction to Python: Lists, Tuples and Loops

## Python Lists

- Python *list* is an ordered, mutable array of objects. A list is constructed by specifying the objects, separated by commas, and enclosed in square brackets.

  `list0 = []` (an empty list)

  `list1 = [0, 1, 2.0, -3, 4/5]` (a list of five numbers)

- Python list can contain references to any type of object: strings, various types of numbers, built-in constants, and even other lists.

  ```
  list2 = ['zero',1, 3.14, 2+3j, True, list1]
  ```

- An item can be retrieved from the list by indexing it (index starts from zero). Try:

  ```
  list1[0], list1[-1], list2[3], list2[5], list2[5][1], list2[5][1][2], list2[0][1]
  ```

- Python lists are *mutable* object. i.e., the items can be reassigned and altered. Try:

  ```
  list2[0] = 0.0; list2
  ```

- List can be *sliced* in the same way as string sequences.

  ```
  list3 = list2[1:4]; list4 = [1::2]; list5 = [::-1]
  ```

# Lists, Tuples and Loops

## List methods

Python lists come with a large number of powerful methods. Since *list* objects are mutable, they can change shapes without having to copy the contents to a new object, as in the strings. Some useful methods are listed below.

- `append(element)`: add an item to the end of the list

  ```
  list3 = list2.append(6)
  ```

- `extend(list)`: add one or more objects by copying them from another list

  ```
  list4 = list3.extend([11,12,13])
  ```

- `insert(index,element)`: insert an item at the specified index

  ```
  list5 = list4.insert(1,100)
  ```

- `pop()`: remove and return the last element from the list

  ```
  list6=list5.pop()
  ```

- `remove(element)`: remove a specified item from the list

  ```
  list2; list2.remove(1)
  ```

## Lists, Tuples and Loops

**List methods:**

- `reverse()`: reverse the list in place

  ```
  list1; list1.reverse(); list1
  ```

- `sort()`: sort the list in place

  ```
  list1.sort(); list1
  ```

- `sorted(list)`: returns a new sorted list, keeping the original unaltered, default is *ascending order*

  ```
  sorted(list4); list4; sorted(list4, reverse=True)
  ```

- Try the following construction of the data structure known as *stack*:

  ```
  stack=[]
  stack.append[a]; stack.append[b]; stack.append[c];stack.append[d]
  print(stack)
  stack.pop()
  print(stack)
  ```

- The string method, *split* generates a list of substrings from a given string, split on a specified separator (default is "white space").

  ```
  s='Sun,Mon,Tue,Wed,Thu,Fri,Sat'

  l=s.split(','); print(l)
  ```

## Lists, Tuples and Loops

### Python Tuples

- Python tuple objects can be thought of *immutable list*. Tuples are created by placing items inside parentheses separated by commas.

  `t0=()` an empty tuple !

  `t1=('one',)` a singleton tuple, note the trailing comma !

  `t2=('one', 2, ['a', 'b', 'c'], 'four', 5.0)`

- Tuples can be indexed or sliced the same way as lists.

  `t2[0]; t2[2][2]; t2[1:3]`

- As tuples are *immutable*, they cannot be appended or extended, or have some elements removed. But we can alter the elements within the list inside the tuples. i.e.,

  `t2[0]='two'; t2[2]=['d', 'e', 'f']` not possible !

  `t2[2][0]='d'; t2[2][1]='e'; t2[2][2]='f'` possible !

- *Tuple packing/unpacking, swapping*

  `t3 = 1, 2, 3` t3=(1,2,3)

  `a, b, c = 1, 2, 3` a=1, b=2, c=3

  `a, b = b, a` t=a, a=b, b=t

- Also try the built-in functions: `list('string'); tuple([1,2,'three'])`

## Lists, Tuples and Loops

### `for` **loops**

- If you want to loop over the elements in a list, you can use a simple `for` loop construction, `for` *item* `in` *list*`:` to loop over the elements in *list*, set to item each step.

```
Math_Spring19 = ['MA305', 'MA345', 'MA432', 'MA438', 'MA448', 'MA453', 'MA484']
for course in Math_Spring19:        #note the colon (:)  at the end
    print(course)                   #note the indentation, four spaces are commonly used
```

- Python has a useful method `range` of referring to a sequence of numbers to loop over. It can be constructed with up to three arguments: `range([a0=0], n, [stride=1])`, where the first integer `a0` and the `stride` (which can be negative) are optional with default values 0 and 1 respectively.

```
n=len(Math_Spring19)
for i in range (n):
    print(i, ':' Math_Spring19[i])
```

- The object created by `range` can be indexed, cast into lists and tuple and iterated over. Try:

```
range(5)[2]; range(1,6)[3]; list(range(0,6,2)); list(range(10,0,-2)); tuple(range(5))
```

## Lists, Tuples and Loops

### `for` loops

- `enumerate` takes an iterate object and produces, for each item in turn, a tuple `(count, item )`, consisting of a counting index and the item itself.

```
Math_Spring19 = ['MA305', 'MA345', 'MA432', 'MA438', 'MA448', 'MA453', 'MA484']
for i, course in enumerate(Math_Spring19):
    print(i, ':' course)
```

- Python's built-in function `zip` creates an iterate object in which each item is a tuple of items taken in turn from the sequences passed to it. This allows us to iterate over many sequences simultaneously.

```
list1=[1, 2, 3, 4]
list2=['a','b','c','d']
for pair in zip(list1,list2):
    print(pair)
```

- The function `zip` can also be used to *unzip* sequences of tuples.

```
list1=[0]
list2=[1, 2]
list3=['a','b','c']
z=zip(list1,list2,list3)
for item in zip(*z):
    print(item)
```

# Lec 10. Introduction to Python: Control Flow

## `if/then/else` statements

- Python controls the decisions that a computer can make in a similar way as other programming languages, through `if/then/else` statements and loop structures with specified termination criteria. Levels of control flow are determined by the indentation level of the code. Four spaces are recommended.

```
if logical_expression_1:   #parenthesis not needed for logical_expression, must end with :
   statements_1            #must be indented, 4 spaces recommended
elif logical_expression_2:
   statements_2
else:
    statements_3
```

- The `if` statements use standard comparison expressions such as $<, >, <=, >=, ==$, and $! =$ to test a truth value. They also incorporate more natural language such as: `is`, `is not`, `and`, `or`.
- A value is `True` unless it is 0 `(int)`, 0.0 `(float)`, empty string, `''`, empty list, `[]`, empty tuple, `()`, or `None` (a special value). Try:

```
for x in range(10):
    if x % 2:     # if x%2=1 (True), x is odd
        print(x, 'is odd!')
    else:         # else x%2=0 (False), x is even
        print(x, 'is even!')
```

# Control Flow in Python

## `while` loops

- The `while` loop is used to execute statements as long as some condition holds.

```
i=0
while(i<10):
    i +=1
    print('Hello ', i)
print()
```

- Try the following example that implements *Euclid's algorithm for the greatest common divisor* of *a* and *b*:

```
a, b = 112, 24
print('a=', a, 'b=', b)
while b:                # the loop is executed till b=0 ('False')
    a, b = b, a%b       # 1) temp=a%b,  2) a=b, 3) b=temp
print('greatest common divisor of a and b:', a)
```

- Python provides some commands (`break/continue/else`) for controlling the flow of a program.

## Control Flow in Python

- The command `break` issued inside a loop immediately ends that loop and moves execution to the statements following the loop.

```
i=0
while True:
    i += 1
    if not (x %15 or x%25):
        break
print(x, 'is divisible by both 15 and 25')
```

- The command `continue` immediately forces the next iteration of the loop without completing the statement block for the current iteration.

```
for i in range91,11):
    if i % 2:
        continue
    print(i, 'is even!')
```

- A `for` or `while` loop may be followed by an `else` block of statements, which will be executed only if the loop is terminated normally.

```
list1=[ 0, 1, 3, -4, 5, 6]
for i, item in enumrate(list1):
    if item < 0:
        break
    else:   # this else will be useful if list 1 does not have negative items
        print(item , 'occurs at index', i)
```

## Next

**Classwork 3.**

**Lists, Loops and Control Flow**

**Due: 09/27/2018**