Collections and Strings

Lists

Dictionaries

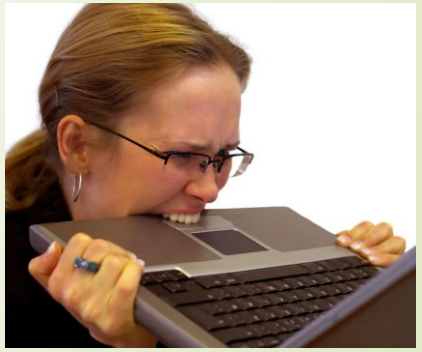
Sets

Strings

Collections: Motivation

- Programs work with simple values: integers, floats, Booleans, strings
- This is inconvenient
  - Suppose a company has three employees
    - Their salaries can be stored in three integer variables: Emp1_sal, Emp2_sal, Emp3_sal
    - If update (increase everyone's salary by 100) is needed, can be done with a simple code: Emp1_sal += 100, Emp2_sal += 100, Emp3_sal +=100
  - Suppose a company has 1000 employees
    - Emp1_sal += 100, Emp2_sal += 100,…, Emp1000_sal +=100
    - If Emp_sal is a collection of 1000 integers, we can access them in a uniform way
    - for i in range(1,1001):

      Emp_sal[i] += 100



Objects, Methods, Mutation

- Some new terminology

Objects and Functions

- An object consists of some data (describing the state of the object) and some operations on that data (which may change the state of the object)
  - Lists, sets, tuples, and dictionaries are kinds of objects
- A function is an operation that is not part of an object
  - A function may take objects as arguments, or produce them as results, but it isn't part of those objects
  - Syntax: function_name(arg, ..., arg)
  - len(my_list) returns the number of elements in my_list

Methods

- A method is an operation that "belongs to" an object
  - You must specify which object you are "talking to"
  - Syntax: object. method_name (arg, ..., arg)
  - Example: my_list.sort() sorts my_list
- Several list methods are given in your resources
  - You can add (append) and remove (pop) items, search, sort, and reverse
  - Remember, Methods can Modify objects (sometimes they do and sometimes don't)

```
my_list = ['one', 'two']
bigger_list = my_list.append('three')
print(bigger_list)
>>None
```

Mutable and Immutable Objects

- Mutating method: a method which changes the state of an object who calls it, e.g.,

```
my_list = ['one', 'two']
my_list.append('three')
print(my_list)
>> ['one', 'two', 'three']
```

  - That's why assignment in the previous slide does not make sense
- Non-mutating method: a method which doesn't change state of an object
  - Those methods produce (other) objects as return values

```
my_set = {'one', 'two', 'tree'}                #we discuss sets in detail later
new_set = my_set.difference({'one', 'two'})
```

- Sometimes there is a mutating and non-mutating method for the same action

```
my_set.difference_update({'one', 'two'})
print(my_set)                # Output: {'three'}
```

- Objects whose all methods are non-mutating are immutable, otherwise, mutable

Lists

Lists: Basics

- Collection of 0 or more elements
  - my_list = ["one", "two", "three"]
  - Lists work best if all the elements are of the same type
    - lst = [1, "two", 3.14] is possible but not desirable
- To access a single element in a list, give the list, then in brackets give the index of the desired element, starting from zero
  - my_list[0] is "one"
  - my_list[1] is "two"
  - my_list[2] is "three"
- You can also use negative indices
  - my_list[-1] is "three"
  - my_list[-2] is "two"
  - my_list[-3] is "one"
- It is an error to index my_list with an integer outside the range -3 to 2
- Lists are mutable, e.g., my_list[0] = "zero"

Processing Lists by Individuals

- Looping with indices

```
for index in range(0, len(my_list)):
    print("Element", index, "is", my_list[index])
```

- Looping without indices
  - If you don't need to know the index of every list element that you process, it's better to use the form of for loop that accesses the list elements directly

```
for element in my_list:
    print(element, "is in the list")
```

Processing Lists as a Whole

- You can get a slice of a list using the syntax list[from : upto]
  - this is a new list containing the elements starting at from and going up to (but not including) upto

```
numbers = [0.0, 1.0, 2.0, 3.0, 4.0, 5.0]
numbers[2:4]                    >> [2.0, 3.0]
numbers[:4]                     >> [0.0, 1.0, 2.0, 3.0]
numbers[2:]                     >> [2.0, 3.0, 4.0, 5.0]
numbers[3:6]                    >> [3.0, 4.0, 5.0]
(numbers[2:])[1] or numbers[2:][1]   >> result?
```

- Other operations: sort, reverse, merge with another list (extend)

Lists of objects

- Objects stored in the list don't have to be simple (integers, strings, etc.)
- They can be lists (nested lists) or other collections
- Typical example: matrix/double array

```
7 5 3 1
9 8 6 4
5 0 8 3
mtrx = [ [7,5,3,1], [9,8,6,4], [5,0,8,3] ]
print(mtrx[2][1])                    #result?
>> 0
```

Sets

Sets: Basic Examples

- s1 = {1, 3.6, 'a', True}
- s2 = set() #creates empty set, can't use {} for this purpose
- s2.add(4) # s2 becomes {4}
- s2.add(1) # s2 becomes {1,4}
- s3 = s1 + s2 # s3 is a union of s1 and s2, note that 1 is not duplicated
  # s3 is {1, 3.6, 'a', True, 4}
- s3.remove('a') # s3 becomes {1, 3.6, 4}

Sets: Key Facts

- Sets are mutable – s2.add(4)
- Sets are unordered
  - I.e., there is no index associated to an element and s[n] doesn't make sense
- Sets do not contain duplicates
- Sets must to store immutable objects:
  - E.g., integers are OK
  - Lists are not OK
  - Sets are not OK
- Iterating through sets

```
for elem in s1:
    print(elem)
```

Dictionaries and Tuples

Dictionaries: Basic Examples

- Dictionaries associate keys with values
- phonebook= {"John Dow" : 123456789, "Michael Jackson" : 987654321}
- d1 = {} #empty dictionary
- print(phonebook["John Dow"])

  >> 123456789
- print(phonebook["David Bowie"])

  >> Error
- phonebook["David Bowie"] = 918273645 #creates a new pair key : value
- phonebook["John Dow"] =6758493021 #updates value for the key
- "Michael Jackson" in phonebook

  >> True #checks if key is present
- del phonebook["Michael Jackson"] #deletes key:value for given key

  "Michael Jackson" in phonebook

  >> False

Dictionaries: Key Facts

- Dictionaries are mutable - phonebook["David Bowie"] = 918273645
- Keys are immutable
  - Integers are OK, e.g., dict[5]
  - Lists are not OK, dict[["David", "Bowie"]]
- Values either mutable or immutable
- Iterating through dictionaries:

```
for key in dict:
    print(key, dict[key])
```

  - Alternative:

```
for key, val in dict.items():
    print(key, val)
```

Tuples

- When you have a small (and better fixed) number of objects
- triple1 = (1, "is smaller than", 5)
- Tuples are ordered and indexed

  print(triple1[2])

  >> 5
- Tuples are immutable

  triple1[2] = 6

  >> Error
  - If need to update, create a new tuple

    triple2 = (triple1[0], triple1[1], 6)
  - Or fully reassign the existing one

    triple1 = (triple1[0], triple1[1], 6)
- Iteration through tuples as through lists (for loop)
- Parentheses can be omitted when defining tuples

  triple2 = triple1[0], triple1[1], 6

Tuples (cont)

- Objects stored in tuples can be mutable (remember: tuples themselves are immutable)
- tup = (‘a’, ‘b’, [100,200])
  - tup[2] = [500, 600] - Error
  - tup[2][0] = 500 - Ok
  - tup[2][1] = 600 - Ok

Strings

Strings: Key Facts

- It makes sense to think of strings as of lists of alphanumerical symbols

```
str = "abcd01"
print(str[0], str[1])
>> a b
print(str[0:4])
>> abcd
```

- However, strings are immutable (similarly to tuples)

```
str[0]='x'
>> Error
```

  - If update required, create a new string

```
new_str = 'x' + str[1:]          # + is a straightforward concatenation operator
str = 'x' + str[1:]              # OK, reassigning the string
```

Strings: Basic Examples

- There is a large number of methods available for working with strings
- Here are some examples:

```
str=“ abc cde \n”
str1 = str.upper())  # str1 is “ ABC CDE \n”
str1 = str.rstrip()      # str1 is “ abc cde”
wds = str1.split()     # wds is a list [“abc”, “cde”] of words
…
```

- You will learn them all as you work and read Python documentation

Food for thought

- We said that strings are immutable but what about
  - str = str.upper()
  - Is it a mutation?
- Note that reassignment is not considered a mutation

  x = x+1

  my_set = my_set.difference({'one', 'two'})

  - It is because the result of x+1 or my_set.difference({'one', 'two'}) is a new object, we only reuse the same variable for it