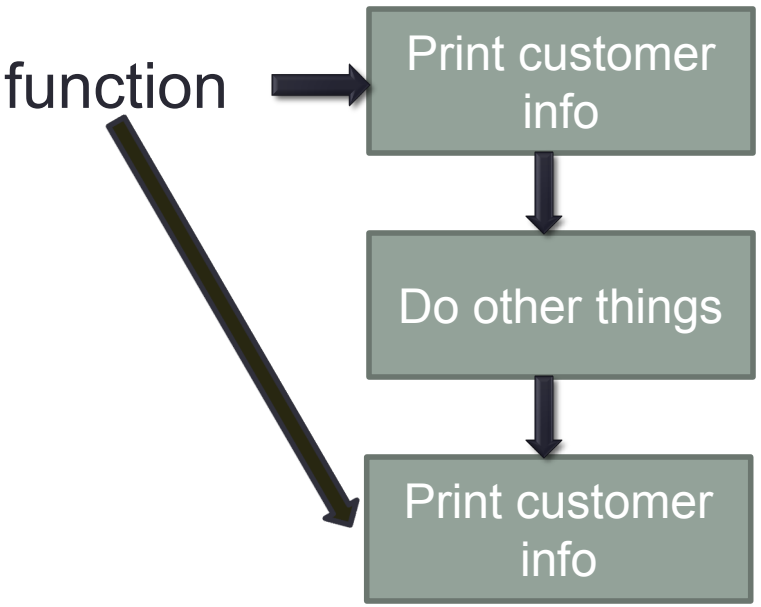
PYTHON REVIEW-PART 2 (FUNCTIONS, STRINGS & DATA STRUCTURES)

Jahan Ghofraniha, Ph.D.

Adapted from Python for Everybody by Charles R. Severance www.py4e.com

Functions

- If you need to repeat a set of python code over and over, it makes sense to construct a function with that code and call the function every time you need it.
- The main purpose of a function is to store code that can be reused.



Functions

- There are two types of functions in Python: built-in and custom.
- The built-in functions are the ones defined inside Python language such as max(), min(), ….
- The custom functions are the ones that the user develops.
- A function has two stages, a function definition stage and a function call or invocation stage.
- We define a function by using keyword def.

```
# function definition
def sum(a,b):
    s = a + b
    return s
```

- Every function at the definition stage has input parameters (i.e. a and b in above example) and a return value, in this case s.
- A function can have multiple returned values.

Function Call

- The function definition is just a definition and does not do anything, i.e. does not get executed until it gets called or invoked.
- The second stage is to call the function;

```
dayhours = 10
nighthours = 5
# invoke function sum and assign the return value to totalhours
totalhours = sum(dayhours, nighthours)
```

- In the above code, we define three variables, dayhours and nighthours and totalhours where the returned value of the function sum gets assigned to totalhours.
- dayhours and nighthours variables are called the function arguments.
- Arguments are the name of the actual parameters that get passed to the function at the time the function gets called.
- We need both stages (definition and invocation) for a function to work.

Buit-in functions

- We saw examples of max() and min as built-in functions.
- Other examples are int(), float() and str() that we use for type conversions.
- When we use int(5/3), we are actually calling a function to convert the output of 5/3 division to an integer.
- The build-in functions normally show up in different colors in IDEs such as Spyder. However, when in doubt if your custom function may end up having the same name as the built-in function make your function name unique such as mysum() or calcpay().

Function Summary

- If you need to repeat few lines of code many times turn that code into a function and call it as many times as required.
- Function usage has two stages: function definition (def sum(….)) and function invocation/calling the function (totalpay = sum(a,b))

```
# function definition
def sum(a,b):
  s = a + b
  return s
# function call
first = 5
second = 10.5
add2nums = sum(first, second)
print(f'Addition of {first} and {second} results in: {add2nums}')
```

Void Function

- It is possible that a function does not return anything. An example would be when you print the results from within the function.

```
def sum(a, b):
    c = a + b
    print(f'the sum is {c}')
```

- When a function does not have a return value it is called a void function.

Exercise

- Ex1: Rewrite the pay computation with time-and-a-half for over-time and create a function called computepay which takes two parameters (hours and rate) and returns the pay.
- Ex2: Write a function that takes a numerical grade as an input parameter and returns a letter grade that corresponds to the numerical grade. Run the program several times and observe the output.

Strings

- A string is a sequence of individual character.
- There are a couple of ways to define a string but the most common way is to define the string by single or double quotes.
- We can refer to the elements of the string by using the square bracket operator.

```
fruit = 'banana'
print(fruit[0])
```

- The index in the square bracket should always be an integer since it is referring to the position of the character.
- Naturally if we try to index a character beyond the end of the string we will get an error.

Strings

- We can traverse a string using a loop:

```
index = 0
while index < len(fruit):
    letter = fruit[index]
    print(letter)
    index = index + 1
```

- The function len() is used to avoid indexing beyond the end of the string.
- String slice is a section of the string. Selecting a slice is similar to selecting a character.

```
fruit = 'banana'
fruit[:3] #from start up to element 3, excluding 3
fruit[3:]
```

Strings

- Strings are immutable (a constant that cannot be changed).

```
fruit = 'banana'
# if we try to change the first element, we get an error
fruit[0] = 'j' # this is because strings are immutable
```

- The in operator checks the presence of a character or set of characters in a string.

```
fruit = 'banana'
'a' in fruit   # you should get True for this line
'he' in fruit  # you should get False for this line
```

- Strings have a series of functions called methods. In the object oriented lecture you will get a good understanding what methods are but for now think of it of series of functions that apply only to strings that can be invoked by stringName.method() format.

String Methods

- In this case stringName is the name of the string variable and .method() is the function.

```
fruit = 'Banana'
# here we use the lower() method to convert all to lower case
Print(fruit.lower())
```

- You can print out all the available methods for strings by using:

```
dir(fruit)
```

String Format

- There are at least four different ways to format a string.
- We will look at the one known as f-string which is valid in 3.6+. It is the easiest and the most intuitive.

```
fruit1 = ‘banana’
fruit2 = ‘strawberry’
print(f’ I love  eating {fruit1} and {fruit2} together’)
```

- For example, when you use the dir() function to find out what methods are available for variable fruit, you can use the f-string format to get a more flexible print out.

```
fruit = ‘banana’
x = dir(fruit)
print(f’Number of methods available for string {fruit1) is {len(x)}’)
```

Exercise

- Ex1: Define a string variable called s1 and assign a value of ‘spam’ to it. Use a loop to print s1 backwards.

- Ex2: Use the sum function from before and use it inside the f-string format operator to change the sum by changing the input parameters. Use 1, 2 first and then use 10, 20. You should see a different print out using f-string format without changing the format statement.

List (Data Structures)

- List is an advanced data structure (a more complex data type than the normal such as int, float, ….) and consists of a sequence of values.
- In its basic form, it can hold different data types and can be indexed similar to a string.
- There are two ways of defining a string, using a square bracket notation or using the list() operator.

```
# using the list operator (list class)
list1 = list() # Create an empty list
list2 = list([2, 3, 4]) # Create a list with elements 2, 3, 4
list3 = list(["red", "green", "blue"]) # Create a list with strings
list4 = list(range(3, 6)) # Create a list with elements 3, 4, 5
list5 = list("abcd") # Create a list with characters a, b, c
```

Lists

```
list1 = [] # Same as list()
list2 = [2, 3, 4] # Same as list([2, 3, 4])
list3 = ["red", "green"] # Same as
                         #list(["red", "green"])
```

- Combining different types in lists:

```
['spam', 2.0, 5, [10, 20]]
```

- Assignment:

```
cheeses = ['Cheddar', 'Edam', 'Gouda']
numbers = [17, 123]
empty = []
print(cheeses, numbers, empty)
```

List is Mutable

- List elements can be changed. Remember we could not do this with strings. String elements after they have been defined cannot be changed.

```
numbers = [17, 123]
numbers[1] = 5
print(numbers)
```

- The most common way to traverse the elements of a list is with a for loop similar to a string.

```
# traversing a list with in-place updating
for i in range(len(numbers)):
    numbers[i] = numbers[i] * 2
```

List Operators

- All the operators for lists have been overloaded (use the same operator syntax but means something else)
- Addition (join two lists together/concatenate):

```
a = [1, 2, 3]
b = [4, 5, 6]
c = a + b
print(c)
[1, 2, 3, 4, 5, 6]
```

- Multiplication (reproduce the list n times):

```
[0] * 4
[0, 0, 0, 0]
[1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

List Slicing

- Similar concept to string slicing. Slicing means selecting a segment of the list.

```
t = ['a', 'b', 'c', 'd', 'e', 'f']
t[1:3]
['b', 'c‘]
```

List Methods:

```
t = ['a', 'b', 'c']
t.append('d')
print(t)
['a', 'b', 'c', 'd']
t2 = ['e', 'f']
t1.extend(t2)
```

Sorting a List

```
print(t1)
['a', 'b', 'c', 'd', 'e', 'f']
# sorting
t = ['d', 'c', 'e', 'b', 'a']
t.sort()
print(t)
['a', 'b', 'c', 'd', 'e']
```

Deleting Elements of a List:

```
t = ['a', 'b', 'c']
x = t.pop(1)
print(t)
['a', 'c']
print(x)
b
```

Deleting Elements of a List

```
t = ['a', 'b', 'c']
del t[1]
print(t)
['a', 'c']
# Use the remove method if don't know the index
t = ['a', 'b', 'c']
t.remove('b')
print(t)
['a', 'c']
```

Tuples (Data Structures)

- Tuples are like lists except they are immutable. Once they are created, their contents cannot be changed.
- If the contents of a list in your application do not change, you should use a tuple to prevent data from being modified accidentally. Furthermore, tuples are more efficient than lists.

```
t1 = () # Create an empty tuple
t2 = (1, 3, 5) # Create a set with three elements
# Create a tuple from a list
t3 = tuple([2 * x for x in range(1, 5)])
# Create a tuple from a string
t4 = tuple("abac") # t4 is ['a', 'b', 'a', 'c']
```

Tuples

- Tuples can be used like lists except they are immutable.

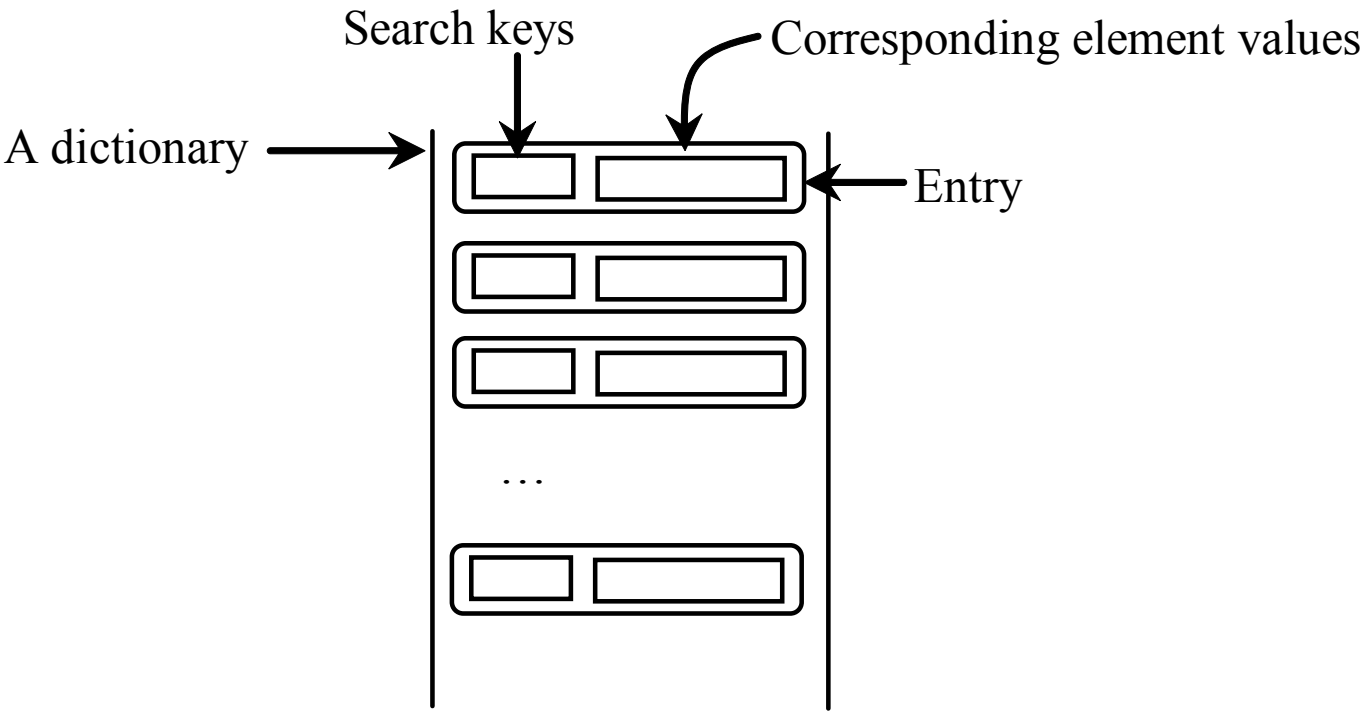
```
tuple1 = ("green", "red", "blue") # Create a tuple
print(tuple1)
tuple2 = tuple([7, 1, 2, 23, 4, 5]) # Create a tuple from a list
print(tuple2)
print("length is", len(tuple2)) # Use function len
print("max is", max(tuple2)) # Use max
print("min is", min(tuple2)) # Use min
print("sum is", sum(tuple2)) # Use sum
print("The first element is", tuple2[0]) # Use indexer
tuple3 = tuple1 + tuple2 # Combine 2 tuples
print(tuple3)
tuple3 = 2 * tuple1 # Multiple a tuple
print(tuple3)
print(tuple2[2 : 4]) # Slicing operator
print(tuple1[-1])
print(2 in tuple2) # in operator
for v in tuple1:
    print(v, end = " ")
print()
```

Dictionaries (Data Structures)

Why dictionary?

Suppose your program stores a million students and frequently searches for a student using the social security number. An efficient data structure for this task is the *dictionary*. A dictionary is a collection that stores the elements along with the keys. The keys are like an indexer.

Dictionaries



Creating Dictionaries

```
dictionary = {} # Create an empty dictionary
dictionary = {"john":40, "peter":45} # Create a dictionary
```

To add an entry to a dictionary, use

```
dictionary[key] = value
```

For example,

```
dictionary["susan"] = 50
```

Looping over Dictionaries

```
for key in dictionary:
    print(key + ":" + str(dictionary[key]))
```

- len(dictionary) returns the number of the elements in the dictionary.

```
dictionary = {"john":40, "peter":45}
"john" in dictionary
True
"johnson" in dictionary
False
```

Dictionary Methods

dict	
keys(): tuple	Returns a sequence of keys.
values(): tuple	Returns a sequence of values.
items(): tuple	Returns a sequence of tuples (key, value).
clear(): void	Deletes all entries.
get(key): value	Returns the value for the key.
pop(key): value	Removes the entry for the key and returns its value.
popitem(): tuple	Returns a randomly-selected key/value pair as a tuple and removes the selected entry.

Dictionary Example

```
d = {'a':10, 'b':1, 'c':22}
mylist = list()
for key, val in d.items() :
    mylist.append( (val, key) )
```

Exercise

- Ex1: Rewrite the program that prompts the user for a list of numbers and prints out the maximum and minimum of the numbers at the end when the user enters “done”. Write the program to store the numbers the user enters in a list and use the max() and min() functions to compute the maximum and minimum numbers after the loop completes.
- Ex2: Use the dictionary example in the slides as a starting point and sort the dictionary based on the values.