Python Part 3 - Functions

|  |  |
| --- | --- |
| **Data Control - Functions**   * functions are defined with the **def** statement * no data types for parameters * no data type for returned value | >>> # function sumList sums the contents of its argument  def sumList (lis):  sum = 0  for x in lis:  sum += x  return sum  nums = [10, 20, 30]  sum = sumList(nums)  print("sum is", sum)  sum is 60 |
| **Data Control - Returning None or Nothing**   * If a return is needed to return a null value, return None * If a function doesn't need to return anything, but a condition causes a need to return, simply return | >>> # function printList prints the contents of its argument  # if it encounters a negative value, it returns  def printList (lis):  for x in lis:  if x < 0:  return  print (x)    >>> nums = [10, 20, 30, -25, 50]  >>> printList(nums)  10  20  30 |
| * functions can return multiple values without requiring a surrounding object | >>> # minMax returns two functional values  def minMaxGrades (gradeM):  min = 200 # large value  max = 0 # small value  for grade in gradeM:  if grade < min:  min = grade  if grade > max:  max = grade  return min, max # returns 2 values  >>> grades = [100, 90, 95, 75]  >>> low, high = minMaxGrades(grades)  >>> print ("low is", low, "high is", high)  low is 75 high is 100 |
| **Data Control - Scope**   * uses **static scoping with nesting** * variables assigned a value outside of functions are considered **global** * since Python doesn't have declarations, a local assignment in a function makes a variable a local throughout that function (even before that assignment) * if a variable is referenced which doesn't have a local assignment, the reference is to either a nonlocal or a global variable | >>> # assigning a value to a variable makes it a local to the function  def funcA():  y = 50 # assignment makes this a local  funcB()  print("A: x =", x, "y =", y)  def funcB():  print("B: x =", x, "y =", y) # referenced the global  x = 10  y = 20  funcA()  print("Outside: x =", x, "y =", y)  **Output:**  ?? |
| **Data Control - Global Statement**   * within a function to assign a value to a global variable, it can be made a global assignment by using a **global *variableName*** statement. | >>> # global statement makes the variable a global, allowing it to be assigned  def funcA():  global y  y = 50  funcB()  print("A: x =", x, "y =", y)  def funcB():  print("B: x =", x, "y =", y)  x = 10  y = 20  funcA()  print("Outside: x =", x, "y =", y)  **Output:**  ?? |
| What happens in this code?  Note that x and y in funcB are being incremented by 5. Is there a problem? | >>> # The compound assignments make x and y local to funcB  def funcA():  y = 50  funcB()  print("A: x =", x, "y =", y)  def funcB():  **x += 5**  **y += 5**  print("B: x =", x, "y =", y)  x = 10  y = 20  funcA()  print("outside: x =", x, "y =", y)  **Output?**  UnboundLocalError: local variable 'x' referenced before assignment |
| **Data Control - Nonlocals**   * Since Python uses static-scoping, nonlocal references will be to surrounding function or globals * To force Python to treat a variable reference as a nonlocal (typically because you want to assign a value to it), we use a **nonlocal *variableName*** statement | >>> # nested funcB references nonlocal x in funcA  def funcA():  def funcB():  print("B: x =", x, "y =", y)  x = 11  funcB()  print("A: x =", x, "y =", y)  x = 10  y = 20  funcA()  **Output:**  ?? |
|  | >>> # Specifying nonlocal and global to allow variable assignment  def funcA():  def funcB():  nonlocal x  global y  x += 5  y += 5  print("B: x =", x, "y =", y)  x = 11  y = 21  funcB()  print("A: x =", x, "y =", y)  x = 10  y = 20  funcA()  **Output:**  ?? |
| **Parameter Passing**   * Uses **by value object reference;** it copies a reference to the object * Therefore, the contents of the object can be changed causing those changes to be known to the corresponding argument | >>> # modify the contents of the object  def change (lis):  lis[1] = "APPLE"  fruits = ["GRAPE", "BANANA", "ORANGE"]  change(fruits)  >>> print(fruits)  ['GRAPE', 'APPLE', 'ORANGE'] |
| * Instead of modifying the object, suppose we change the value of the parameter; this changes what the parameter is pointing to -- not the argument | >>> # change the parameter for a list, not affecting the argument  def change2 (lis):  lis = ["APPLE" "PEAR"]  fruits = ["GRAPE", "BANANA", "ORANGE"]  change2(fruits)  >>> print(fruits)  ['GRAPE', 'BANANA', 'ORANGE'] |
| * Numbers are treated as object references; each number can be a different object | >>> # change the parameter for a number  def changeNum(numx):  numX = 15  print("changeNum: ", numX)  numA = 10  changeNum(numA)  print("after changeNum: ", numA)  **Output:**  changeNum: 15  after changeNum: 10 |
| **Optional Arguments**  Python provides a mechanism to supply default values if an argument is not provided.  **def**  *funcName*(*parmName1*, *parmName2, ...*  *optParm1=defaultValue1, ...*)   * The function can have required parameters and optional parameters. * Optional parameters are provided a default value. * Arguments can be listed in any order when using the *argName=value* form | >>> # define a function which provides default values for marital status  # and exemption  def setW2(ssn, exemption=1,marital="S"):  print ("ssn=", ssn, "exemption=", exemption, "marital=", marital)  setW2("111-11-1111")  setW2("222-22-2222", 4, "M")  setW2("333-33-3333", marital="M")  setW2("444-44-4444", marital="S", exemption=3)  setW2(exemption=2, ssn="555-55-5555")  **Output:**  ssn= 111-11-1111 exemption= 1 marital= S  ssn= 222-22-2222 exemption= 4 marital= M  ssn= 333-33-3333 exemption= 1 marital= M  ssn= 444-44-4444 exemption= 3 marital= S  ssn= 555-55-5555 exemption= 2 marital= S |
| **Variable Number of Arguments**  In addition to optional arguments, Python supports a mechanism to provide a variable number of arguments. .  **def**  *funcName*(*parmName1*, *parmName2, ...*  *\*remainingParm*)   * After the last required parameter, specify an \* before the next parameter. * This will tell the translator to place the remaining arguments in a tuple which is referenced by the name of *remainingParm.* | >>> # define a function which returns a count of numeric values > a specified value  # It also makes certain each value is an int or a float  def countGreater (atLeast, \*remValues):  count = 0  for x in remValues:  print (x, type(x))  if type(x) in (int, float):  if x > atLeast:  count += 1  return count  print (" count greater is "  , countGreater(59, 65, 40, 85.0, "Marco", 90, "Polo"))  **Output:**  65 <class 'int'>  40 <class 'int'>  85.0 <class 'float'>  Marco <class 'str'>  90 <class 'int'>  Polo <class 'str'>  count greater is 3 |
| **Variable Number of Arguments continued**  **def**  *funcName*(*parmName1*, *parmName2, ...*  *\*\*remainingParm*)   * After the last required parameter, specify an \*\* before the next parameter. * This will tell the translator to place the remaining arguments in a dictionary which is referenced by the name of *remainingParm.* * The variable arguments should be in the *argName=value* form | >>> # define a function which is passed a variable number of arguments  # in the argName=value form  def showTags(id, \*\*properties):  print ("Properties for ", id)  for tag, value in properties.items():  print(tag, value)  showTags("978-15932799999", Title="Don't Get Bit by Python"  , Pages=600, InvType="BOOK", Price=30.99)  showTags("B00IR8H55A", size=3.2, Price=99.99, Color="SILVER")  **Output:**  Properties for 978-15932799999  Title Don't Get Bit by Python  InvType BOOK  Price 30.989999999999998  Pages 600  Properties for B00IR8H55A  Color SILVER  Price 99.989999999999995  size 3.2000000000000002 |
|  |  |
|  |  |
|  |  |