**Mine Best of Code Warriors Conversion**

|  |  |  |
| --- | --- | --- |
| def getNumber(number):  fizzbuzz = ""  if isinstance(number, int):  if (number % 5 == 0) and (number % 3 == 0):  fizzbuzz = "BOTH"  elif number % 3 == 0:  fizzbuzz = "THREE"  elif number % 5 == 0:  fizzbuzz = "FIVE"  else:  fizzbuzz = number  return fizzbuzz  else:  return "Not a valid integer."  def getNumberRange(first, last):  a\_list = []  if isinstance(first or last, int):  if first < last:  for i in range(first, last + 1):  a\_list.append(getNumber(i))  return a\_list  else:  for i in range(first, last - 1, -1):  a\_list.append(getNumber(i))  return a\_list  else:  return "Integer/s not valid." | def getNumber(n):  t = n % 3 == 0  f = n % 5 == 0  return 'BOTH' if t and f else 'THREE' if t else 'FIVE' if f else n  def getNumberRange(first, last):  sign = **cmp**(last, 0)  return **map**(getNumber, range(first, last + sign, sign)) | *Part 1: def getNumber(n)*  Basic if statement (**ternary operator**):  Many programming languages have a ternary operator, which define a conditional expression. The most common usage is to make a terse simple conditional assignment statement. In other words, it offers one-line code to evaluate the first expression if the condition is true, otherwise it evaluates the second expression.  **[return][expression1][if][condition][else][expression2][if][condtion2]….**  *Part2: def getNumberRange(first, last)*  **cmp(x, y)**: Compares two objects x and y and returns:  1 if x > y; -1 if x < y; 0 if x == y  **map(function, iterable)**: Apply function to every item of iterable and **return a list** of the results. Point of difference between map and filter function is map will return the result of the function thereby returning a list of ouput with type determined by by the function. Filter will return a list containing output determined by the list). In this CW example, map() returns the output from getNumber()  **range(start, stop[,step])**: creates a list containing arithmetic progressions most often used in for loops.  return map([run getNumber function and apply through the iterable – a list defined by range with variable sign allowing both positive and negative integer progression. |
| def disemvowel(string):  new\_string = ""  vowels = "aeiou"  for i in string:  if i.lower() not in vowels:  new\_string += i  return new\_string | def disemvowel(s):  return s.translate(None, "aeiouAEIOU")  def disemvowel(string):  return "".join(c for c in string if c.lower() not in "aeiou") | **str.translate(table[, deletechars])**: **returns the string** where all characters occurring in the optional argument *deletechars* are removed and the remaining characters have been mapped through the given translation table.  Table example:  in = “aeiou”  out = ”12345”  trans = maketrans(in, out)  “Hello”.translate(trans) 🡪 “H2ll4”  **str.join(iterable)**: **Return a string** which is the concatenation of the strings in the iterable ‘iterable’. The separator between elements is the string proving this method.  join each letter in ‘string’ separated by “” i.e. each successive letter ONLY IF they are not in ‘aeiou’. This is an example of a ternary operator (simplified conditional statement):  **[expression: join where sep. is “”]applied to [iterable: c for c in string][condition: if … not in …..]** |
| def is\_prime(n):  x = 0  if isinstance(n, int) and n > 1:  for i in range(2, n - 1):  if n % i == 0:  x += 1  break  return True if x == 0 else False  else:  return False | def is\_prime(n):  return n > 1 and all(n % i for i in xrange(2, n))  def is\_prime(n):  return n > 1 and all(n % i for i in xrange(int(n \*\* 0.5), 1, -1))  def is\_prime(n):  'Return True if n is a prime number otherwise return False'  return n > 1 and all(n % i for i in range(2, n)) | **all(iterable): return a Boolean**, tru if all elements of the iterable are true (or if the iterable is empty).  Equivalent to:  *def all(iterable):*  *for element in iterable:*  *if not element:*  *return False*  *return True* |
| def hamming(a, b):  count = 0  for i in range(len(a)):  if a[i] != b[i]:  count += 1  return count | def hamming(a,b):  return sum(ch1 != ch2 for ch1, ch2 in zip(a, b))  def hamming(a,b):  return [ca == cb for ca, cb in zip(a, b)].count(False)  def hamming(a,b):  return sum([1 for x, y in zip(a,b) if x != y])  def hamming(a, b):  return sum(map(lambda x: not x[0] == x[1], zip(a, b)))  def hamming(a,b):  return sum(a[i] != b[i] for i in range(len(a)))  def hamming(a,b):  return len([idx for idx in xrange(len(a)) if a[idx] != b[idx]]) | **sum(iterable[, start]):** sums optional *start* and the items of an *iterable* from left to right and returns the toal.  **zip([iterable, …]):** Returns a list of tuples (the same as lists except you can’t change their values and they are assigned values with ‘(‘ not ‘[‘), where the i-th tuple contains the i-th element from each of the argument sequences or iterables.  Example:  >>> x = [1, 2, 3]  >>> y = [4, 5, 6]  >>> zipped = zip(x, y)  >>> zipped  [(1, 4), (2, 5), (3, 6)]  def hamming(a,b):  return sum(ch1 != ch2 for ch1, ch2 in zip(a, b))  In English:   * zip creates a list of tuple, each containing a letter in sequence from the strings a and b. * ‘for ch1, ch2 in ‘ is the iteration statement applied to the zipped list of tuples. * ‘ch1!=ch2’ is the function applied to the list.   **return sum([function] [iterable] [list])** |
| def sum(\*args):  total = 0  for i in args:  if isinstance(i, int):  total += i  return total | def sum(\*args):  return \_\_builtins\_\_.sum(filter(lambda x: type(x) == int, args))  def sum(\*args):  return reduce(lambda x,y: x + y, filter(lambda n: type(n) == int, args))  def sum(\*args):  return reduce(lambda acum,x: acum+x if isinstance(x,(int,long)) else acum, args,0)  def sum(\*args):  args2 = [a for a in args if type(a) == int]  return reduce(lambda x,y: x + y, args2)  ssum = sum  def sum(\*args):  return ssum([x for x in args if type(x) is int])  sum = (lambda s: lambda \*args: s(i for i in args if isinstance(i, int)))(sum)  def sum(\*args):  return reduce(lambda memo, x: memo + x, [a for a in args if isinstance(a, (int, long))], 0) | Highlighted solution is in the form of:  **Expression | Iterable | Conditional Statement** |
| def solution(nums):  try:  return sorted(nums)  except TypeError:  return [] | def solution(nums):  return sorted(nums) if nums else []  solution = lambda l: sorted(l) if l else [] |  |
| def search\_names(logins):  no\_good = []  for l in logins:  if l[0][-1] == '\_':  no\_good.append(l)  return no\_good  (not my solution but similar in style) | def search\_names(logins):  return filter(lambda x: x[0].endswith(“\_”), logins) | **filter(function, iterable)**: returns a list from those elements in iterable for which function returns True. If interable is a string or a tuple, filter returns a string or a tuple.  return a list of the elements of list ‘logins’ where the lambda anon fx equates to True. |
| def identify\_weapon(character):  weapons = {"Laval": "Shado Valious", "Cragger": "Vengdualize", "Lagravis": "Blazeprowlor",  "Crominus": "Grandorius", "Tormak": "Tygafyre", "LiElla": "Roarburn"}  try:  newlist = filter(lambda x: character in x, weapons)  return newlist[0]+"-"+weapons[newlist[0]]  except:  return "Not a character" | def identify\_weapon(character):  weapons = { 'Laval' : 'Shado Valious', 'Cragger' : 'Vengdualize', 'Lagravis' : 'Blazeprowlor', 'Crominus' : 'Grandorius', 'Tormak' : 'Tygafyre', 'LiElla' : 'Roarburn' }  return '%s-%s' % (character, weapons[character]) if character in weapons else 'Not a character'  def identify\_weapon(character):  tbl = {"Laval" : "Laval-Shado Valious", "Cragger" : "Cragger-Vengdualize", "Lagravis" : "Lagravis-Blazeprowlor", "Crominus" : "Crominus-Grandorius", "Tormak" : "Tormak-Tygafyre", "LiElla" : "LiElla-Roarburn"}  return tbl.get(character, "Not a character") | First solution by mortonfox which is so perfect it makes me weep. So basic you don’t gain anything by analysing it.  Second uses the dict type operation *dict.***get(key[, default])**, which returns the value if key is in the dict, else default, if default is not given, None is returned. Their solution bends the rules slightly by iterating through a dictionary that has a preformatted value in the form x: x-y instead of x: y |
| def triangle\_type(a, b, c):  tri = sorted([a, b, c])  if tri[0] + tri[1] <= tri[2]:  return 0  elif tri[0]\*\*2 + tri[1]\*\*2 > tri[2]\*\*2:  return 1  elif tri[0]\*\*2 + tri[1]\*\*2 == tri[2]\*\*2:  return 2  else:  return 3 | def triangle\_type(a, b, c):  min\_, avg\_, max\_ = sorted([a, b, c])    if min\_ + avg\_ <= max\_:  return 0    return cmp(max\_\*\*2, min\_\*\*2 + avg\_\*\*2) + 2 | Use of the cmp method was pretty clever.  Depending on the value of max, min and avg squared cmp returns -1, 0 or 1 then +2 is added and result returned. (avg should be median) |
| def convert\_temp(temp, from\_scale, to\_scale):  if from\_scale == "C":  x = temp  elif from\_scale == "F":  x = ((temp - 32) \* 5 / 9)  elif from\_scale == "K":  x = (temp - 273.15)  elif from\_scale == "R":  x = ((temp - 491.67) \* 5 / 9)  elif from\_scale == "De":  x = (100 - temp \* 2 / 3)  elif from\_scale == "N":  x = (temp \* 100 / 44)  elif from\_scale == "Re":  x = (temp \* 5 / 4)  elif from\_scale == "Ro":  x = ((temp - 7.5) \* 40 / 21)  if to\_scale == "C":  return x  elif to\_scale == "F":  return int(x \* 9 /5 + 32)  elif to\_scale == "K":  return int(x + 273.15)  elif to\_scale == "R":  return int((x + 273.15) \* 9 / 5)  elif to\_scale == "De":  return int((100 - x) \* 3 / 2)  elif to\_scale == "N":  return int(x \* 33 / 100)  elif to\_scale == "Re":  return int(x \* 4 / 5)  elif to\_scale == "Ro":  return int(x \* 21 / 40 + 7.5) | TO\_KELVIN = {  'C': (1, 273.15),  'F': (5.0 / 9, 459.67 \* 5.0 / 9),  'R': (5.0 / 9, 0),  'De': (-2.0 / 3, 373.15),  'N': (100.0 / 33, 273.15),  'Re': (5.0 / 4, 273.15),  'Ro': (40.0 / 21, -7.5 \* 40 / 21 + 273.15),  }  def convert\_temp(temp, from\_scale, to\_scale):  if from\_scale == to\_scale:  return temp  if from\_scale != 'K':  (a, b) = TO\_KELVIN[from\_scale]  temp = a \* temp + b  if to\_scale == 'K':  return int(round(temp))  (a, b) = TO\_KELVIN[to\_scale]  return int(round((temp - b) / a)) | While I am fine with a conditional solution, the best of others has a cool use of a dictionary.  Firstly they create the dictionary of from\_scale keys and values consisting of a list a, b of the conversion formula arranged in a way that they can all be applied the same way by calling later on – a \* temp + b.  Then check dictionary for from scale key, assign list values to a and b, evaluate the function to get temp.  Check dictionary againg for to\_scale, rearrangement of conversion formula is useful here again as it’s rearranged for (temp – b )/ a to give temp. |
|  |  |  |