Python Data Science Toolbox (P1)

Writing Your Own Functions

1. docstrings
    - will describe what your function does
    - serve as documentation of your function
    - placed in the immediate line after the function header
    - write in triple quotation marks (''' ''')

```
#function with one parameter and return one value
def square(value):
  ''' Return the square of a value'''
  new_value = values ** 2
  return new_value
```

2. multiple parameters and return values
    - function with multiple parameters

```
def raise_to_power(value1, value2):
  """Raise value1 to the power of value2."""
  new_value = value1 ** value2
  return new_valueCallfunction:
```

    - return multiple values

```
def raise_both(value1, value2):
  """Raise value1 to the power of value2 and vice versa."""
  new_value1 = value1 ** value2
  new_value2 = value2 ** value1
  new_tuple = (new_value1, new_value2)
  return new_tuple

result = raise_both(2, 3)
print(result)

#Output:
(8, 9)
```

Default Arguments, Variabale-length Arguments and Scope

1. scope
    - part of the program where an object or name may be accessible
    - 3 types
        1. global scope - defined in the main body of a script
        2. local scope - defined inside a function (cannot access the variable outside the function)
        3. built-in scope - names in the pre-defined built-ins module

- global scope vs. local scope

```
new _value = 10
def square(value):
  ''' Return the square of a value'''
  new_value = values ** 2
  return new_value

suqare(3)
#Output:
9

print(new_value)
#Output:
10

'''If the local scope cannot be found, the global scope will be searched'''

#Alter the value of a global name within a function call
new _val = 10
def square(value):
  ''' Return the square of a value'''
  global new_val
  new_val = values ** 2
  return new_val

square(3)
#Output:
100

print(new_val)
#Output:
100
```

2. nested functions

```
#Type 1
def mod2plus5(x1, x2, x3):
  """Returns the remainder plus 5 of three values."""

  def inner(x):
  """Returns the remainder plus 5 of a value."""
  return x % 2 + 5

return (inner(x1), inner(x2), inner(x3))


print(mod2plus5(1, 2, 3))

#Output:
(6, 5, 6)


#Type 2
def raise_val(n):
  """Return the inner function."""

  def inner(x):
  """Raise x to the power of n."""
  raised = x ** n
  return raised

return inner

square = raise_val(2)
cube = raise_val(3)

print(square(2), cube(4))

#Output:
4 64
```

```
#Type 3 (using nonlocal)
def outer():
  """Prints the value of n."""
  n = 1

  def inner():nonlocal n
    n = 2
    print(n)

inner()
print(n)

outer()

#Output:
#the result is 2 because it alter the value of n in inner function and also enclosing scope which is outer function
2
2
```

3. sequence of scope searched (LEGB)
    1. local scope
    2. enclosing functions
    3. global scope
    4. built-in functions

4. function with default argument

```
#pow = 1 is the default argument
def power(number, pow=1):
  """Raise number to the power of pow."""
  new_value = number ** pow
  return new_value

power(9, 2)

#Output:
81

#Output:
power(9, 1)

#output:
9

power(9)

#Output:
9
```

5. function with flexible arguments
    - *args (can be used if not sure how many arguments needed to be pass) - tuple

```
def add_all(*args):
  """Sum all values in *args together."""

  # Initialize sum
  sum_all = 0

  # Accumulate the sum
```

```
  for num in args:
    sum_all += num

  return sum_all

add_all(1)

#Output:
1

add_all(1, 2)

#Output:
3

add_all(5, 10, 15, 20)

#Output:
50
```

- **kwargs (arguments proceeded by identifiers) - dictionary

```
def print_all(**kwargs):
  """Print out key-value pairs in **kwargs."""

  # Print out the key-value pairs
  for key, value in kwargs.items():
    print(key + \": \" + value)

print_all(name="dumbledore", job="headmaster")

#Output:
job: headmaster
name: dumbledore
```

Lambda Functions and Error-handling

1. lambda is keyword that allow the user to write the function in a quicker way (no recommended all the time)

```
raise_to_power = lambda x, y: x ** y

raise_to_power(2, 3)

#Output:
8
```

2. anonymous functions
   - function map takes two arguments : map(func, seq)
   - map() applies the function to all elements in the sequence
   - able to pass lambda function without specify the name into the map function and applies to elements which known as anonymous function

```
nums = [48, 6, 9, 21, 1]

square_all = map(lambda num: num ** 2, nums)

#need to print the results in the list else it will only print the object id
print(list(square_all))
```

```
#Output:
[2340, 36, 81, 441, 1]
```

- filter() is used to filter out the elements from a list that do not satisfy certain criteria
- reduce() is used to return a single value as a result but it is needed to be imported from the functools module

3. error handling
   - exceptions - caught error during execution by using try-except clause

```
def sqrt(x):
  """Returns the square root of a number."""
  try:
    return x ** 0.5
  except:
    print('x must be an int or float')

sqrt(2)

#Output:
2.0

sqrt(10.0)

#Output:
3.1622776601683795

sqrt('hi)

#Output:
x must be an int or float
```

   - raise error instead of catch error only'

```
def sqrt(x):
  """Returns the square root of a number."""
  if x < 0:
    raise ValueError('x must be non-negative')
  try:
    return x ** 0.5
  except TypeError:
    print('x must be an int or float')
```