Note)

Flexible arguments: *args (1)

add_all.py

```
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
```

Flexible arguments: **kwargs

kwargs.py

```
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)
```

```
In [1]: print_all(name="dumbledore", job="headmaster")
job: headmaster
name: dumbledore
```

1)

Map() and lambda functions

So far, you've used lambda functions to write short, simple functions as well as to redefine functions with simple functionality. The best use case for lambda functions, however, are for when you want these simple functionalities to be anonymously embedded within larger expressions. What that means is that the functionality is not stored in the environment, unlike a function defined with `def`. To understand this idea better, you will use a lambda function in the context of the `map()` function. Recall from the video that `map()` applies a function over an object, such as a list. Here, you can use lambda functions to define the function that `map()` will use to process the object. For example:

```
nums = [2, 4, 6, 8, 10]
```

```
result = map(lambda a: a ** 2, nums)
```

You can see here that a lambda function, which raises a value `a` to the power of 2, is passed to `map()` alongside a list of numbers, `nums`. The *map object* that results from the call to `map()` is stored in `result`. You will now practice the use of lambda functions with `map()`. For this exercise, you will map the functionality of the `add_bangs()` function you defined in previous exercises over a list of strings.

- In the `map()` call, pass a lambda function that concatenates the string `'!!!'` to a string `item`; also pass the list of strings, `spells`. Assign the resulting map object to `shout_spells`.
- Convert `shout_spells` to a list and print out the list.

```
# Create a list of strings: spells
spells = ["protego", "accio", "expecto patronum", "legilimens"]

# Use map() to apply a lambda function over spells: shout_spells
shout_spells = map(lambda a:a+'!!!',spells)

# Convert shout_spells to a list: shout_spells_list
shout_spells_list = list(shout_spells)

# Convert shout_spells into a list and print it
print(shout_spells_list)
```

```
['protego!!!', 'accio!!!', 'expecto patronum!!!', 'legilimens!!!']
```

2)

```
# Create a list of strings: fellowship
fellowship = ['frodo', 'samwise', 'merry', 'aragorn', 'legolas', 'boromir', 'gimli']

# Use filter() to apply a lambda function over fellowship: result
result = filter(lambda a:len(a)>6 , fellowship)
print (result)
```

```
<filter object at 0x7fd9353dc278>
```

```
# Convert result to a list: result_list
result_list = list(result)

# Convert result into a list and print it
print(result_list)
```

```
['samwise', 'aragorn', 'legolas', 'boromir']
```

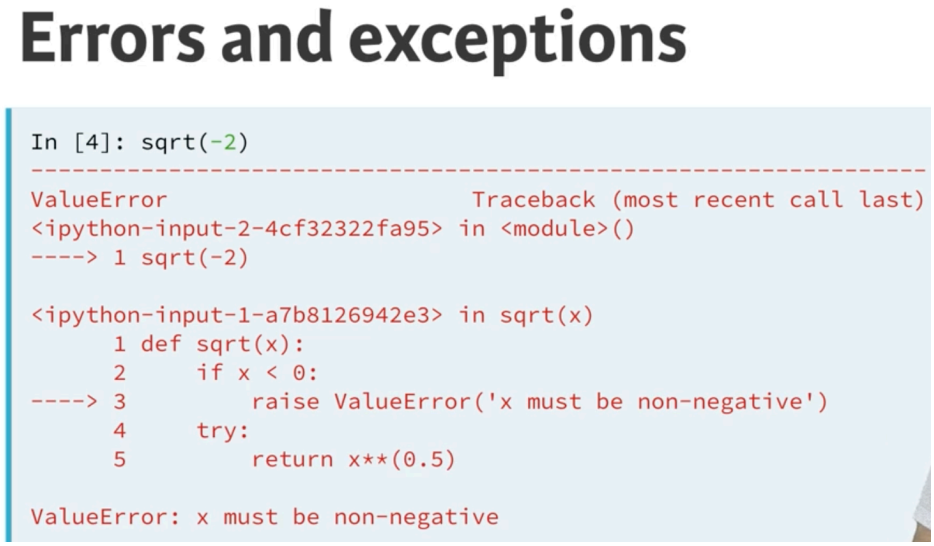
3)

```
# Import reduce from functools
from functools import reduce
```

```
# Create a list of strings: stark
stark = ['robb', 'sansa', 'arya', 'eddard', 'jon']

# Use reduce() to apply a lambda function over stark: result
result = reduce(lambda item1,item2:item1+item2, stark)

# Print the result
print(result)
```

robbsansaaryaeddardjon

Note)



Toolbox2

Chapter 1)

Note)

Iterating over iterables: next()

```
In [1]: word = 'Da'

In [2]: it = iter(word)

In [3]: next(it)
Out[3]: 'D'

In [4]: next(it)
Out[4]: 'a'

In [5]: next(it)
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-11-2cdb14c0d4d6> in <module>()
----> 1 next(it)
StopIteration:
```

Iterating at once with *

```
In [1]: word = 'Data'

In [2]: it = iter(word)

In [3]: print(*it)
D a t a

In [4]: print(*it)
```

← No more values to go through!

Using enumerate()

```
In [1]: avengers = ['hawkeye', 'iron man', 'thor', 'quicksilver']

In [2]: e = enumerate(avengers)

In [3]: print(type(e))
<class 'enumerate'>

In [4]: e_list = list(e)

In [5]: print(e_list)
[(0, 'hawkeye'), (1, 'iron man'), (2, 'thor'), (3, 'quicksilver')]
```

enumerate() and unpack

```
In [1]: avengers = ['hawkeye', 'iron man', 'thor', 'quicksilver']

In [2]: for index, value in enumerate(avengers):
  ....:     print(index, value)
0 hawkeye
1 iron man
2 thor
3 quicksilver

In [3]: for index, value in enumerate(avengers, start=10):
  ....:     print(index, value)
10 hawkeye
11 iron man
12 thor
13 quicksil
```

Using zip()

```
In [1]: avengers = ['hawkeye', 'iron man', 'thor', 'quicksilver']

In [2]: names = ['barton', 'stark', 'odinson', 'maximoff']

In [3]: z = zip(avengers, names)

In [4]: print(type(z))
<class 'zip'>

In [5]: z_list = list(z)

In [6]: print(z_list)
[('hawkeye', 'barton'), ('iron man', 'stark'), ('thor',
'odinson'), ('quicksilver', 'maximoff')]
```

```
In [4]: print(*z)
('hawkeye', 'barton') ('iron man', 'stark') ('thor', 'odinson')
('quicksilver', 'maximoff')
```

zip() and unpack

```
In [1]: avengers = ['hawkeye', 'iron man', 'thor', 'quicksilver']

In [2]: names = ['barton', 'stark', 'odinson', 'maximoff']

In [3]: for z1, z2 in zip(avengers, names):
  ....:     print(z1, z2)
hawkeye barton
iron man stark
thor odinson
quicksilver maximoff
```

1)

# Create a zip object from mutants and powers: z1
z1 = zip(mutants,powers)

# Print the tuples in z1 by unpacking with *
print(*z1)

```
# Re-create a zip object from mutants and powers: z1
z1 = zip(mutants,powers)

# 'Unzip' the tuples in z1 by unpacking with * and zip(): result1, result2
result1, result2 = zip(*z1)

# Check if unpacked tuples are equivalent to original tuples
print(result1 == mutants)
print(result2 == powers)
```
True
True

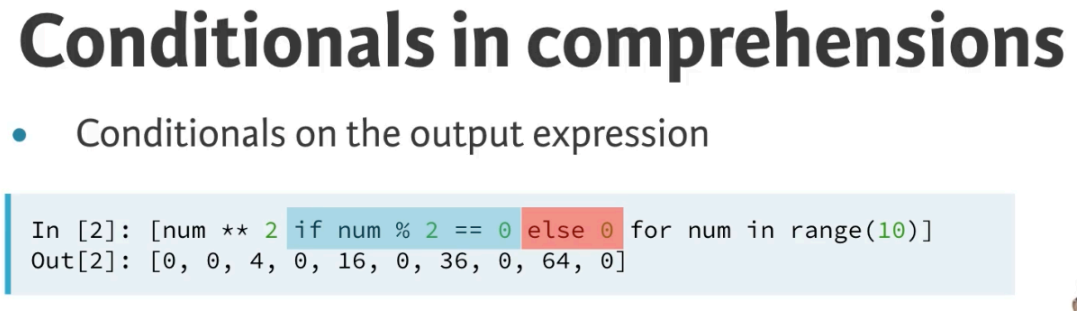
Note)

2)

```
# Initialize an empty dictionary: counts_dict
counts_dict = {}

# Iterate over the file chunk by chunk
for chunk in pd.read_csv('tweets.csv',chunksize=10):

    # Iterate over the column in DataFrame
    for entry in chunk['lang']:
        if entry in counts_dict.keys():
            counts_dict[entry] += 1
        else:
            counts_dict[entry] = 1

# Print the populated dictionary
print(counts_dict)
```

{'et': 1, 'und': 2, 'en': 97}

Note)

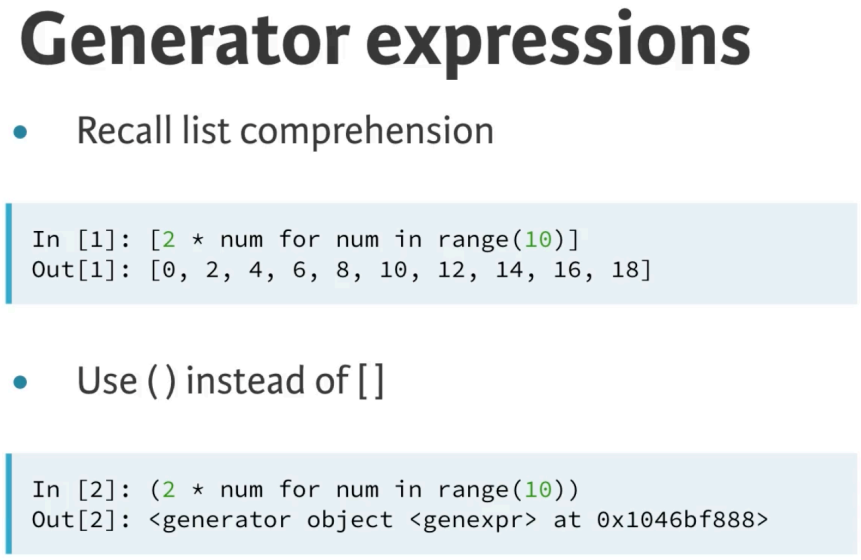


3)

```
# Create a list of strings: fellowship
fellowship = ['frodo', 'samwise', 'merry', 'aragorn', 'legolas', 'boromir', 'gimli']

# Create list comprehension: new_fellowship
new_fellowship = [member if len(member)>=7 else '' for member in fellowship]

# Print the new list
print(new_fellowship)
```

['', 'samwise', '', 'aragorn', 'legolas', 'boromir', '']

Note)



Printing values from generators

```
In [1]: result = (num for num in range(6))

In [2]: print(next(result))
0

In [3]: print(next(result))
1

In [4]: print(next(result))
2

In [5]: print(next(result))
3

In [6]: print(next(result))
4
```

Lazy evaluation

Build a generator function

sequence.py

```
def num_sequence(n):
    """Generate values from 0 to n."""
    i = 0
    while i < n:
        yield i
        i += 1
```

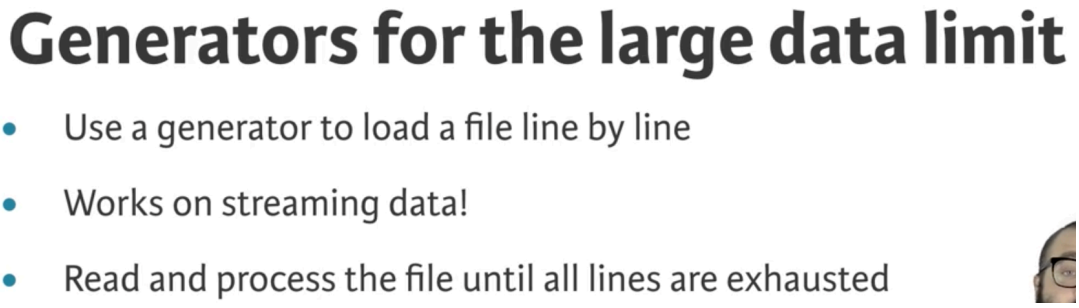
Chapter 3)

1)

```
# Define lists2dict()
def lists2dict(list1, list2):
    """Return a dictionary where list1 provides
    the keys and list2 provides the values."""

    # Zip lists: zipped_lists
    zipped_lists = zip(list1, list2)

    # Create a dictionary: rs_dict
    rs_dict = dict(zipped_lists)

    # Return the dictionary
    return rs_dict

# Call lists2dict: rs_fxn
rs_fxn = lists2dict(feature_names,row_vals)
```

```
# Print rs_fxn
print(rs_fxn)
```

Note)



2)

```
# Open a connection to the file
with open('world_dev_ind.csv') as file:

    # Skip the column names
    file.readline()

    # Initialize an empty dictionary: counts_dict
    counts_dict = {}

    # Process only the first 1000 rows
    for j in range(1000):

        # Split the current line into a list: line
        line = file.readline().split(',')

        # Get the value for the first column: first_col
        first_col = line[0]

        # If the column value is in the dict, increment its value
        if first_col in counts_dict.keys():
            counts_dict[first_col] += 1

        # Else, add to the dict and set value to 1
        else:
            counts_dict[first_col] = 1

# Print the resulting dictionary
print(counts_dict)
```

{'Euro area': 119, 'European Union': 116, 'Caribbean small states': 77, 'East Asia & Pacific (developing only)': 123, 'Heavily indebted poor countries (HIPC)': 18, 'Central Europe and the Baltics': 71, 'Fragile and conflict affected situations': 76, 'Europe & Central Asia (developing only)': 89, 'East Asia & Pacific (all income levels)': 122, 'Europe & Central Asia (all income levels)': 109, 'Arab World': 80}

3)

```
# Define read_large_file()
def read_large_file(file_object):
    """A generator function to read a large file lazily."""

    # Loop indefinitely until the end of the file
    while True:

        # Read a line from the file: data
        data = file_object.readline()

        # Break if this is the end of the file
        if not data:
            break

        # Yield the line of data
        yield data

# Open a connection to the file
with open('world_dev_ind.csv') as file:

    # Create a generator object for the file: gen_file
    gen_file = read_large_file(file)

    # Print the first three lines of the file
    print(next(gen_file))
    print(next(gen_file))
    print(next(gen_file))
```

CountryName,CountryCode,IndicatorName,IndicatorCode,Year,Value

Arab World,ARB,"Adolescent fertility rate (births per 1,000 women ages 15-19)",SP.ADO.TFRT,1960,133.56090740552298

Arab World,ARB,Age dependency ratio (% of working-age population),SP.POP.DPND,1960,87.7976011532547

4)
Writing a generator to load data in chunks (3)

Great! You've just created a generator function that you can use to help you process large files.
Now let's use your generator function to process the World Bank dataset like you did previously. You will process the file line by line, to create a dictionary of the counts of how many times each country appears in a column in the dataset. For this exercise, however, you won't process just 1000 rows of data, you'll process the entire dataset!
The generator function `read_large_file()` and the csv file `'world_dev_ind.csv'` are preloaded and ready for your use. Go for it!

```
# Initialize an empty dictionary: counts_dict
counts_dict = {}

# Open a connection to the file
with open('world_dev_ind.csv') as file:

    # Iterate over the generator from read_large_file()
    for line in read_large_file(file):

        row = line.split(',')
        first_col = row[0]

        if first_col in counts_dict.keys():
            counts_dict[first_col] += 1
        else:
            counts_dict[first_col] = 1

# Print
print(counts_dict)
```

5)
Writing an iterator to load data in chunks (1)

Another way to read data too large to store in memory in chunks is to read the file in as DataFrames of a certain length, say, 100. For example, with the pandas package (imported as `pd`), you can do `pd.read_csv(filename, chunksize=100)`. This creates an iterable **reader object**, which means that you can use `next()` on it.
In this exercise, you will read a file in small DataFrame chunks with `read_csv()`. You're going to use the World Bank Indicators data `'ind_pop.csv'`, available in

your current directory, to look at the urban population indicator for numerous countries and years.

```
# Import the pandas package
import pandas as pd

# Initialize reader object: df_reader
df_reader = pd.read_csv('ind_pop.csv', chunksize=10)

# Print two chunks
print(next(df_reader))
print(next(df_reader))
```

6)
Writing an iterator to load data in chunks (4)

In the previous exercises, you've only processed the data from the first DataFrame chunk. This time, you will aggregate the results over all the DataFrame chunks in the dataset. This basically means you will be processing the **entire** dataset now. This is neat because you're going to be able to process the entire large dataset by just working on smaller pieces of it!
You're going to use the data from `'ind_pop_data.csv'`, available in your current directory. The packages `pandas` and `matplotlib.pyplot` have been imported as `pd` and `plt` respectively for your use.

```
# Initialize reader object: urb_pop_reader
urb_pop_reader = pd.read_csv('ind_pop_data.csv', chunksize=1000)

# Initialize empty DataFrame: data
data = pd.DataFrame()

# Iterate over each DataFrame chunk
for df_urb_pop in urb_pop_reader:

    # Check out specific country: df_pop_ceb
    df_pop_ceb = df_urb_pop[df_urb_pop['CountryCode'] == 'CEB']

    # Zip DataFrame columns of interest: pops
    pops = zip(df_pop_ceb['Total Population'],
               df_pop_ceb['Urban population (% of total)'])

    # Turn zip object into list: pops_list
    pops_list = list(pops)
```

```
    # Use list comprehension to create new DataFrame column 'Total Urban
Population'
    df_pop_ceb['Total Urban Population'] = [int(tup[0] * tup[1]) for tup in
pops_list]

    # Append DataFrame chunk to data: data
    data = data.append(df_pop_ceb)

# Plot urban population data
data.plot(kind='scatter', x='Year', y='Total Urban Population')
plt.show()
```