

```

In [1]: 1 # This cell obtains United States Consumer Price Index (CPI) data from an online server.
2 # A function (defined in the present cell) that incorporates CPI data
3 # will later be used to make inflation adjustments on movie-related sales and budget figures.
4 #
5 # MICHAEL COLLINS, 2020-09-15_0729_MDT
6
7 import pandas as pd
8 import numpy as np
9 from IPython.display import display
10
11 # Obtain monthly United States consumer price indices from "inflationdata.com"
12 # and store them in pandas dataframe cpi_df.
13 source_url = 'https://inflationdata.com/Inflation/Consumer_Price_Index/HistoricalCPI.aspx?reloaded=true'
14 cpi_df = pd.read_html(source_url)[0].drop(columns='Ave.')
15 cpi_df.set_index('Year', drop=True, inplace=True, verify_integrity=False)
16
17 year_to_cpi = np.round(cpi_df.mean(axis=1, skipna=True), decimals=6)
18
19 def inflation_multiplier(from_year, into_year):
20     cpi_ratio = None
21     bomb = False
22     if not bomb:
23         from_cpi = None
24         try:
25             from_cpi = year_to_cpi[from_year]
26         except:
27             pass
28         if not isinstance(from_cpi, float):
29             bomb = True
30     if not bomb:
31         into_cpi = None
32         try:
33             into_cpi = year_to_cpi[into_year]
34         except:
35             pass
36         if not isinstance(into_cpi, float):
37             bomb = True
38     if not bomb:
39         cpi_ratio = np.round((into_cpi/from_cpi), decimals=6)
40     return cpi_ratio
41
42 check_inflation_multiplier = True
43 if not check_inflation_multiplier:
44     print("SKIPPING quality checks of inflation_multiplier function...")
45     print()
46 else:
47     print("PERFORMING quality checks of inflation_multiplier function...")
48     print()
49
50     print("Annual values of the United States Consumer Price Index are")
51     print("contained in the pandas series 'year_to_cpi', as follows:")
52     display(year_to_cpi)
53
54     print("Here are the inflation adjustment factors one would apply to convert ")
55     print("the value of [Year YYYY dollars] into the value of [Year 2020 dollars]:")
56     print()
57     from_years = list(range(1900,2031))
58     for j in [2020]:
59         for i in from_years:
60             factor_ij = inflation_multiplier(from_year=i, into_year=j)
61             print("inflation_multiplier(" + str(i) + ", " + str(j) + ") = " + repr(factor_ij))
62             print()
63     print()
64
65

```

PERFORMING quality checks of inflation\_multiplier function...

Annual values of the United States Consumer Price Index are  
contained in the pandas series 'year\_to\_cpi', as follows:

Year	
2020	258.045375
2019	255.657417
2018	251.106833
2017	245.119583
2016	240.007167
	...
1917	12.825000
1916	10.883333
1915	10.108333
1914	10.016667

```
In [2]: 1 # This cell obtains film industry data directly from the website "the-numbers.com"
2 # and stores it in a dictionary called "movieHandle_to_movieDoss".
3 #
4 # A cleaned-up version of the above data is stored in another dictionary called
5 # "eventHandle_to_eventDoss". That dictionary forms the basis of a pandas dataframe
6 # in a subsequent cell of this Notebook.
7 #
8 # MICHAEL COLLINS, 2020-09-11_2133_MDT
9 #
10 import datetime
11 from datetime import date
12 import requests
13 from bs4 import BeautifulSoup
14 import random
15 import string
16 from dateutil.parser import parse as dateparse
17 import os
18 import pathlib
19 import numpy as np
20 import pandas as pd
21 import winsound
22 from IPython.display import display
23 import matplotlib.pyplot as plt
24 import seaborn as sns
25 #
26 # Note: TN is an abbreviation for "The Numbers"; shorthand for 'the-numbers dot com'
27 # Note: BOY is an abbreviation for "Box-Office Year"
28 # Note: bgt is an abbreviation for the "budget" table
29 # Note: tgy is an abbreviation for the "top-grossing yearly" table
30 #
31 #
32 # Constants related to processing of generic calendar dates
33 YEAR_PLAUSIBLE_FIRST = 1900
34 YEAR_PLAUSIBLE_LAST = 2100
35 DATE_PLAUSIBLE_FIRST = datetime.date(YEAR_PLAUSIBLE_FIRST,1,1)
36 DATE_PLAUSIBLE_LAST = datetime.date(YEAR_PLAUSIBLE_LAST,12,31)
37 WEEKDAY_MONDAY = 0
38 WEEKDAY_TUESDAY = 1
39 WEEKDAY_WEDNESDAY = 2
40 WEEKDAY_THURSDAY = 3
41 WEEKDAY_FRIDAY = 4
42 WEEKDAY_SATURDAY = 5
43 WEEKDAY_SUNDAY = 6
44 #
45 # Constants related to "the-numbers.com" website
46 TN_SALES OMIT = [",", "$"]
47 TN_SEATS OMIT = [","]
48 TN_MAIN_URL = "https://the-numbers.com"
49 TN_PARENT_FOLDER = "./dayduh"
50 TN_MAIN_FOLDER = "./dayduh/the-numbers"
51 TN_TOPGROSS_SUBFOLDER = "/top-gross"
52 TN_TOPGROSS_SUBURL = "/market/"
53 TN_TOPGROSS_URL_SUFFIX = "/top-grossing-movies"
54 TN_BUDGETS_SUBFOLDER = "/budgets"
55 TN_BUDGETS_SUBURL = "/movie/budgets/"
56 TN_MOVIE_SUBFOLDER = "/movie"
57 TN_MOVIE_SUBURL = "/movie/"
58 TN_MOVIE_URL_SUFFIX = "#tab=summary"
59 TN_DISTRIB_SUBURL = "/market/distributor/"
60 TN_GENRE_SUBURL = "/market/genre/"
61 TN_YEAR_CONSIDER_FIRST = 1977
62 TN_YEAR_CONSIDER_LAST = 2020
63 TN_GENRE_HANDLES = ["_unknown_genre_", "Action", "Adventure", "Black-Comedy", "Comedy", "Concert-or-Performance",
64 "Documentary", "Drama", "Horror", "Multiple-Genres", "Musical", "Reality",
65 "Romantic-Comedy", "Thriller-or-Suspense", "Western"]
66 TN_GENRE_ABBRS = ["--none--", "ACTION", "ADVENTURE", "BLK-COMEDY", "COMEDY", "CONCERT",
67 "DOCUMENTARY", "DRAMA", "HORROR", "MULTI-GENRE", "MUSICAL", "REALITY",
68 "ROM-COMEDY", "THRILLER", "WESTERN"]
69 #
70 # dictionaries that relate genre handles to genre abbreviations and vice versa
71 TN_GENRE_INDICES = list(range(len(TN_GENRE_ABBRS)))
72 gHandle_to_gAbbr = dict(zip(TN_GENRE_HANDLES, TN_GENRE_ABBRS))
73 gAbbr_to_gHandle = dict(zip(TN_GENRE_ABBRS, TN_GENRE_HANDLES))
74 gAbbr_to_gIndex = dict(zip(TN_GENRE_ABBRS, TN_GENRE_INDICES))
75 gIndex_to_gAbbr = dict(zip(TN_GENRE_INDICES, TN_GENRE_ABBRS))
```

```

76 # functions that allow commentary to be printed,
77 # depending on value of COMMENTARY_LEVEL
78 comm1 = lambda s: print(s) if COMMENTARY_LEVEL >= 1 else None
79 comm2 = lambda s: print(s) if COMMENTARY_LEVEL >= 2 else None
80 comm3 = lambda s: print(s) if COMMENTARY_LEVEL >= 3 else None
81 comm4 = lambda s: print(s) if COMMENTARY_LEVEL >= 4 else None
82
83 COMMENTARY_LEVEL = 3
84
85 # This function returns the first calendar day of a given Box Office Year
86 def boxOfficeYear_firstDay(boxOfficeYear):
87     # The first day of Box Office Year YYYY is
88     # [the day AFTER the first Sunday in Calendar Year YYYY]
89     d = None
90     bomb = False
91     bomb = bomb or (not isinstance(boxOfficeYear, int))
92     bomb = bomb or (not boxOfficeYear >= YEAR_PLAUSIBLE_FIRST)
93     bomb = bomb or (not boxOfficeYear < YEAR_PLAUSIBLE_LAST)
94     if not bomb:
95         d = datetime.date(boxOfficeYear, 1, 1)
96         while not d.weekday() == WEEKDAY_SUNDAY:
97             d += datetime.timedelta(days=1)
98         while not d.weekday() == WEEKDAY_MONDAY:
99             d += datetime.timedelta(days=1)
100    return d
101
102 # This function returns the Last calendar day of a given Box Office Year
103 def boxOfficeYear_lastDay(boxOfficeYear):
104     # The Last day of Box Office Year YYYY is
105     # [the first Sunday in Calendar Year (YYYY + 1)]
106     d = None
107     bomb = False
108     bomb = bomb or (not isinstance(boxOfficeYear, int))
109     bomb = bomb or (not boxOfficeYear >= YEAR_PLAUSIBLE_FIRST)
110     bomb = bomb or (not boxOfficeYear < YEAR_PLAUSIBLE_LAST)
111     if not bomb:
112         d = datetime.date(boxOfficeYear + 1, 1, 1)
113         while not d.weekday() == WEEKDAY_SUNDAY:
114             d += datetime.timedelta(days=1)
115    return d
116
117 # This function returns the Box Office Year associated with a given calendar day
118 def date_to_boxOfficeYear(d):
119     # The Last day of Box Office Year YYYY is
120     # [the first Sunday in Calendar Year (YYYY + 1)]
121     boxOfficeYear = None
122     bomb = False
123     bomb = bomb or (not isinstance(d, datetime.date))
124     bomb = bomb or (not d >= DATE_PLAUSIBLE_FIRST)
125     bomb = bomb or (not d <= DATE_PLAUSIBLE_LAST)
126     if not bomb:
127         d_calendar_year = d.year
128         mainBOY_firstDay = boxOfficeYear_firstDay(d_calendar_year)
129         bomb = not isinstance(mainBOY_firstDay, datetime.date)
130     if not bomb:
131         if d < mainBOY_firstDay:
132             boxOfficeYear = d_calendar_year - 1
133         else:
134             boxOfficeYear = d_calendar_year
135    return boxOfficeYear
136
137
138 # This function makes two audible beeps.
139 # An audible signal, suitably implemented, can alert the user
140 # when (for example) a long-running computational task is completed.
141 def make_beeps():
142     Freq = 440 # Set Frequency To 440 Hertz
143     Dur = 500 # Set Duration To 500 ms == 0.5 second
144     winsound.Beep(Freq,Dur)
145     Freq = 880 # Set Frequency To 880 Hertz
146     Dur = 500 # Set Duration To 500 ms == 0.5 second
147     winsound.Beep(Freq,Dur)
148
149
150 # extract movie handle from movie href
151 def get_movie_handle(href):

```

```

152     def remove_mh_prefix(v):
153         return v[len(TN_MOVIE_SUBURL):] if v.startswith(TN_MOVIE_SUBURL) else v
154     def remove_mh_suffix(v):
155         return v[:-len(TN_MOVIE_URL_SUFFIX)] if v.endswith(TN_MOVIE_URL_SUFFIX) else v
156     return remove_mh_suffix(remove_mh_prefix(href))
157
158 # extract distributor handle from distributor href
159 def get_distrib_handle(v):
160     return v[len(TN_DISTRIB_SUBURL):] if v.startswith(TN_DISTRIB_SUBURL) else v
161
162 # extract genre handle from genre href
163 def get_genre_handle(v):
164     return v[len(TN_GENRE_SUBURL):] if v.startswith(TN_GENRE_SUBURL) else v
165
166 def tnURL_movieWebsite(movie_handle, as_path=False):
167     bomb = False
168     bomb = bomb or (not isinstance(movie_handle, str))
169     bomb = bomb or (not len(movie_handle) >= 1)
170     f_result = None
171     if not bomb:
172         path_ref = "/" + movie_handle + ".html"
173         url_ref = movie_handle + TN_MOVIE_URL_SUFFIX
174         if as_path:
175             f_result = TN_MAIN_FOLDER + TN_MOVIE_SUBFOLDER + path_ref
176         else:
177             f_result = TN_MAIN_URL + TN_MOVIE_SUBURL + url_ref
178     return f_result
179
180 def tnURL_budgets_glob(start_rank=1, as_path=False):
181     bomb = False
182     bomb = bomb or (not isinstance(start_rank, int))
183     bomb = bomb or (not start_rank >= 1)
184     f_result = None
185     if not bomb:
186         page, j = divmod(start_rank, 100)
187         if page > 0:
188             i_rank = (100 * page) + 1
189             s_rank = str(i_rank).zfill(4)
190             path_ref = "/all_" + s_rank + ".html"
191             url_ref = "all/" + s_rank
192         else:
193             path_ref = "/all_0001.html"
194             url_ref = "all/1"
195         if as_path:
196             f_result = TN_MAIN_FOLDER + TN_BUDGETS_SUBFOLDER + path_ref
197         else:
198             f_result = TN_MAIN_URL + TN_BUDGETS_SUBURL + url_ref
199     return f_result
200
201 def tnURL_topGross_byYear(box_office_year, as_path=False):
202     bomb = False
203     bomb = bomb or (not isinstance(box_office_year, int))
204     bomb = bomb or (not box_office_year >= TN_YEAR_CONSIDER_FIRST)
205     bomb = bomb or (not box_office_year <= TN_YEAR_CONSIDER_LAST)
206     f_result = None
207     if not bomb:
208         path_ref = "/top-grossing-movies_" + str(box_office_year) + ".html"
209         url_ref = str(box_office_year) + "/top-grossing-movies"
210         if as_path:
211             f_result = TN_MAIN_FOLDER + TN_TOPGROSS_SUBFOLDER + path_ref
212         else:
213             f_result = TN_MAIN_URL + TN_TOPGROSS_SUBURL + url_ref
214     return f_result
215
216
217 def local_curated_folder_exists(desired_folder):
218     result = False
219
220     still_going = True
221     while still_going:
222         # Check whether a Locally-curated version of the desired webContent already exists.
223         try:
224             old_folder = pathlib.Path(desired_folder)
225             old_folder_exists = old_folder.exists()
226         except:
227             print("Issue: could not determine whether local folder {" + desired_folder + "} already exists.")

```

```

228         still_going = False
229         break
230     if old_folder_exists:
231         # This is the most-preferred outcome. The desired folder already exists.
232         still_going = False
233         result = True
234         break
235
236     # At this point, the desired Local folder does NOT already exist.
237
238     # Try to create the desired Local folder.
239     try:
240         new_folder = pathlib.Path(desired_folder).mkdir(mode=0o777, parents=True, exist_ok=False)
241     except:
242         print("Issue: could not create new local folder {" + desired_folder + "}.")
243         still_going = False
244         break
245
246     # At this point, we THINK a new instance of the desired Local folder was just created.
247     # Make sure it is actually there.
248     try:
249         new_folder = pathlib.Path(desired_folder)
250         new_folder_exists = new_folder.exists()
251     except:
252         print("Issue: could not determine whether **newly-created** local folder {" + desired_folder + "} already exists.")
253         still_going = False
254         break
255     if new_folder_exists:
256         # This is the second-most-preferred outcome. The desired folder was created.
257         still_going = False
258         result = True
259         break
260
261     return result
262
263
264 def local_curated_file_exists(webContent_local_path, webContent_url):
265     result = False
266
267     still_going = True
268     while still_going:
269
270         # Check whether a locally-curated version of the desired webContent already exists.
271         try:
272             old_file = pathlib.Path(webContent_local_path)
273             old_file_exists = old_file.exists()
274         except:
275             print("Issue: could not determine whether local file {" + webContent_local_path + "} already exists.")
276             still_going = False
277             break
278         if old_file_exists:
279             # This is the most favorable outcome. A Locally curated version of the webContent was found.
280             still_going = False
281             result = True
282             break
283
284         # Download the webContent via webContent_url
285         try:
286             r = requests.get(webContent_url)
287             webContent = r.content
288         except:
289             print("Issue: webContent was not obtained from URL {" + webContent_url + "}.")
290             still_going = False
291             break
292
293         # Save the webContent as a locally-curated file.
294         try:
295             with open(webContent_local_path, 'wb') as f:
296                 f.write(webContent)
297         except:
298             print("Issue: webContent was not written to file {" + webContent_local_path + "}.")
299             still_going = False
300             break
301
302         # Confirm that the (brand-new) Locally-curated file exists
303         new_file_exists = False

```

```

304
305     new_file = pathlib.Path(webContent_local_path)
306     new_file_exists = new_file.exists()
307
308     except:
309         print("Issue: after webContent was stored locally, could not determine " +
310             "whether the local copy {" + webContent_local_path + "} exists.")
311         still_going = False
312         break
313
314     if not new_file_exists:
315         print("Issue: after webContent was stored locally, the local copy {" +
316             webContent_local_path + "} does not exist.")
317         still_going = False
318         break
319
320     # Confirm that the (brand-new) locally-curated file can be read
321     try:
322         with open(webContent_local_path, 'rb') as g:
323             localContent = g.read()
324
325     except:
326         print("Issue: after webContent was stored locally, the local copy {" +
327             webContent_local_path + "} exists, but could not be read.")
328         still_going = False
329         break
330
331     # Confirm that the archived version of the webContent
332     # is exactly the same as the downloaded version of the webContent
333     if not localContent == webContent:
334         print("Issue: after webContent was stored locally, the local version " +
335             "found at {" + webContent_local_path + "} is NOT an exact replica of the " +
336             "webContent that was downloaded from {" + webContent_url + "}.")
337         still_going = False
338         break
339
340     # The archived webContent is an exact replica of the downloaded webContent
341     still_going = False
342     result = True
343
344     print("An exact replica of the webContent at {" + webContent_url +
345         "} was stored in local file {" + webContent_local_path + "}.")
346
347
348
349 # =====
350 # IN THIS SECTION,
351 #     QUALITY CHECKS ON THE FUNCTIONS DEFINED ABOVE
352 #     MAY OPTIONALY BE PERFORMED
353 #     AT THE DISCRETION OF THE USER
354 #     BY SETTING THE "check_????" FLAGS TO True or False, ACCORDINGLY
355 # =====
356
357 check_genre_dicts = True
358 if not check_genre_dicts:
359     print("SKIPPING quality checks of genre-related dictionaries...")
360     print()
361 else:
362     print("PERFORMING quality checks of genre-related dictionaries...")
363     print()
364
365     print("Quality check of gHandle_to_gAbbr...")
366     for i, k in enumerate(TN_GENRE_HANDLES):
367         def check_same(v1, v2):
368             return "PASS" if (v1 == v2) else "FAIL"
369         v_expected = TN_GENRE_ABBRS[i]
370         v_lookup = "--absent--"
371         try:
372             v_lookup = gHandle_to_gAbbr[k]
373         except:
374             v_lookup = None
375         print("i=" + str(i) + ", key=" + repr(k) + ", v_expected=" + repr(v_expected) +
376             ", v_lookup=" + repr(v_lookup) + ", status=" + check_same(v_expected, v_lookup))
377
378     print()
379
380     print("Quality check of gAbbr_to_gHandle...")

```

```

380     for i, k in enumerate(TN_GENRE_ABRBS):
381         def check_same(v1, v2):
382             return "PASS" if (v1 == v2) else "FAIL"
383         v_expected = TN_GENRE_HANDLES[i]
384         v_lookup = "--absent--"
385         try:
386             v_lookup = gAbbr_to_gHandle[k]
387         except:
388             v_lookup = None
389         print("i=" + str(i) + ", key=" + repr(k) + ", v_expected=" + repr(v_expected) +
390             ", v_lookup=" + repr(v_lookup) + ", status=" + check_same(v_expected, v_lookup))
391     print()
392
393 check_boxOfficeYear_date_functions = True
394 if not check_boxOfficeYear_date_functions:
395     print("SKIPPING quality checks of boxOfficeYear-related date functions...")
396     print()
397 else:
398     print("PERFORMING quality checks of boxOfficeYear-related date functions...")
399     print()
400
401     print("Calculating the beginning and ending dates of various Box Office Years...")
402     print()
403     BOYs = [y for y in range(TN_YEAR_CONSIDER_FIRST, TN_YEAR_CONSIDER_LAST + 1)]
404     for BOY in BOYs:
405         day_first = boxOfficeYear_firstDay(BOY)
406         day_last = boxOfficeYear_lastDay(BOY)
407
408         print("BOY = {" + str(BOY) + "}, starts " +
409             day_first.strftime("%a") + " {" +
410             repr(day_first) + "}, ends " +
411             day_last.strftime("%a") + " {" +
412             repr(day_last) + "}")
413     print()
414
415     print("Verifying that the calculated boxOfficeYear for each individual day")
416     print("that occurs DURING that particular boxOfficeYear is in fact the")
417     print("known (prescribed) boxOfficeYear...")
418     print()
419     BOYs = [y for y in range(1915, 2032)]
420     for BOY in BOYs:
421         day_first = boxOfficeYear_firstDay(BOY)
422         day_last = boxOfficeYear_lastDay(BOY)
423         d = day_first
424         BOY_num_days_agree = 0
425         BOY_num_days_disagree = 0
426         BOY_list_days_disagree = []
427         while d <= day_last:
428             calc_BOY = date_to_boxOfficeYear(d)
429             if calc_BOY == BOY:
430                 BOY_num_days_agree += 1
431             else:
432                 BOY_num_days_disagree += 1
433                 BOY_list_days_disagree.append(d)
434
435             d += datetime.timedelta(days=1)
436         if BOY_num_days_disagree > 0:
437             print("BOX OFFICE YEAR " + str(BOY) + ":")
438             print("    " + str(BOY_num_days_disagree) + " days DISAGREE, as follow:")
439             for d in BOY_list_days_disagree:
440                 print("        " + repr(d))
441     print()
442
443 check_tnURL_budgets_glob = False
444 if not check_tnURL_budgets_glob:
445     print("SKIPPING quality checks of function tnURL_budgets_glob...")
446     print()
447 else:
448     print("PERFORMING quality checks of function tnURL_budgets_glob...")
449     print()
450     start_nums = sorted([random.randint(-100, 3001) for k in range(100)])
451     for i in start_nums:
452         print("i = {" + str(i) + "}: ")
453         budgets_glob_path = " + repr(tnURL_budgets_glob(i, as_path=True)))"
454         budgets_glob_url = " + repr(tnURL_budgets_glob(i)))"
455     print()

```

```

456
457 check_tnURL_topGross_byYear = False
458 if not check_tnURL_topGross_byYear:
459     print("SKIPPING quality checks of function tnURL_topGross_byYear...")
460     print()
461 else:
462     print("PERFORMING quality checks of function tnURL_topGross_byYear...")
463     print()
464     box_office_years = [iBOY for iBOY in range(1960,2031)]
465     for iBOY in box_office_years:
466         print("boxOfficeYear = {" + str(iBOY) + "}: ")
467         print("    topGross_byYear_path = " + repr(tnURL_topGross_byYear(iBOY, as_path=True)))
468         print("    topGross_byYear_url = " + repr(tnURL_topGross_byYear(iBOY)))
469     print()
470
471 check_tnURL_movieWebsite = False
472 if not check_tnURL_movieWebsite:
473     print("SKIPPING quality checks of function tnURL_movieWebsite...")
474     print()
475 else:
476     print("PERFORMING quality checks of function tnURL_movieWebsite...")
477     print()
478     # Here are some example movie handles. This is for testing the
479     # operation of the tnURL_movieWebsite function
480     movie_handles = ['(Untitled)', '10', '10-000-B-C', '10-Cloverfield-Lane', '10-Days-in-a-Madhouse',
481                     '10-Questions-for-the-Dalai-Lama', '10-Things-I-Hate-About-You', '10-to-Midnight',
482                     '10-Years', '100-Acres-of-Hell', '100-Arabica', '100-Bloody-Acres', '1001-Grams',
483                     '101-Dalmatians-(1961)', '101-Dalmatians-(1996)', '101-ReykjavA-k', '102-Dalmatians',
484                     '102-Not-Out-(India)', '10E', '10th-and-Wolf', '11-09-01-September-11', '11-11-11',
485                     '11-14', '11th-Hour', '12-(2009-Russian-Federation)', '12-Angry-Men', '12-in-a-Box',
486                     '12-jours-(France-2017)', '12-Monkeys', '12-O-Clock-Boys', '12-Rounds', '12-Strong',
487                     '12-Years-a-Slave', '120-battements-par-minute-(France)-(BPM-Beats-Per-Minute)',
488                     '127-Hours', '127-Hours-(2010)', '13-Going-On-30',
489                     '13-Hours-The-Secret-Soldiers-of-Benghazi', '13-Months-of-Sunshine', '13-Sins',
490                     '13-Tzameti', '13B', '13th-Warrior-The', '1408', '1492-Conquest-of-Paradise',
491                     '15', '15-17-to-Paris-The', '15-fevrier-1839', '15-Minutes', '16-Blocks',
492                     '16-to-Life', '1612', '17-Again', '17-filles-(France)', '1776', '18-Again',
493                     '18-ans-apres', '180-South', '1898-Los-ultimos-de-Filipinas-(Spain)', '1915',
494                     '1917-(2019)', '1941', '1945-(Hungary)', '1969', '1981', '1982',
495                     '1999-Cannes-Intl-Adv-Festival', '2-13', '2-22-(2017)', '2-automnes-3-hivers',
496                     '2-Days-in-New-York', '2-Days-In-The-Valley', '2-Fast-2-Furious', '2-For-the-Money',
497                     '2-Guns', '2-Manner-2-Frauen-4-Probleme', '2-ou-3-choses-que-je-sais-d-elle',
498                     '2-States', '20-centimetros', '20-Dates', '20-Feet-From-Stardom', '200-Cartas',
499                     '200-Cigarettes', '20000-Days-on-Earth', '20000-Leagues-Under-the-Sea-(1916)',
500                     '20000-Leagues-Under-the-Sea-(1954)', '2001-A-Space-Odyssey',
501                     '2005-Academy-Award-Nominated-Short-Films-The',
502                     '2006-Academy-Award-Nominated-Short-Films', '2009-Oscar-Shorts', '2010',
503                     '2010-Oscar-Shorts', '2011-Oscar-Shorts', '2012', '2012-Oscar-Shorts',
504                     '2012-Time-for-Change', '2013-Oscar-Shorts', '2014-Oscar-Shorts', '2015-Oscar-Shorts',
505                     '2016-Obama-s-America']
506     for mh in movie_handles[0:10]:
507         movieWebsite_path = tnURL_movieWebsite(mh, as_path=True)
508         movieWebsite_url = tnURL_movieWebsite(mh)
509
510         print("movie_handle = {" + mh + "}: ")
511         print("    movieWebsite_path = " + repr(movieWebsite_path))
512         print("    movieWebsite_url = " + repr(movieWebsite_url))
513         print()
514
515         # Now store a local version of the movie's web content
516
517         if not local_curated_file_exists(movieWebsite_path, movieWebsite_url):
518             print("Issue: Attempt to store webContent at URL {" +
519                  movieWebsite_url + "} as local file {" +
520                  movieWebsite_path + "} has failed.")
521             print()
522
523
524
525 # =====
526 # IN THIS SECTION,
527 # LOCAL FOLDERS (FOR CACHED LOCAL VERSIONS OF WEB CONTENT) ARE CREATED, AS NEEDED.
528 #
529 # =====
530
531 issues_found = True

```

```

532 if local_curated_folder_exists(TN_PARENT_FOLDER):
533     if local_curated_folder_exists(TN_MAIN_FOLDER):
534         if local_curated_folder_exists(TN_MAIN_FOLDER + TN_MOVIE_SUBFOLDER):
535             if local_curated_folder_exists(TN_MAIN_FOLDER + TN_BUDGETS_SUBFOLDER):
536                 issues_found = False
537             else:
538                 print("Issue: Attempt to create local folder {" +
539                       TN_MAIN_FOLDER + TN_TOPGROSS_SUBFOLDER + "} has failed.")
540             else:
541                 print("Issue: Attempt to create local folder {" +
542                       TN_MAIN_FOLDER + TN_BUDGETS_SUBFOLDER + "} has failed.")
543         else:
544             print("Issue: Attempt to create local folder {" +
545                   TN_MAIN_FOLDER + TN_MOVIE_SUBFOLDER + "} has failed.")
546         else:
547             print("Issue: Attempt to create local folder {" +
548                   TN_MAIN_FOLDER + "} has failed.")
549     else:
550         print("Issue: Attempt to create local folder {" +
551               TN_PARENT_FOLDER + "} has failed.")
552 if issues_found:
553     # Pitch a fit.
554     print("Unable to cache webContent in local data curation folders.")
555     print()
556     print("Purposely Terminating program execution, now.")
557     assert False
558 print("All desired folders for local curation of webContent were verified to be present.")
559 print()
560
561
562
563
564 # =====
565 # IN THIS SECTION,
566 # HARVESTING OF DATA FROM "the-numbers dot com" IS PERFORMED.
567 #
568 # EACH MODE OF DATA HARVEST OPERATION
569 # MAY OPTIONALY BE PERFORMED AT THE DISCRETION OF THE USER
570 # BY SETTING "harvest_???" FLAGS TO True OR False, ACCORDINGLY
571 # =====
572
573 movieHandle_to_movieDoss = dict()
574 distribHandle_to_distribDoss = dict()
575 genreHandle_to_genreDoss = dict()
576
577 bgt_rows_cumulative = 0
578 tgy_rows_cumulative = 0
579
580
581 harvest_bgt = True
582 if not harvest_bgt:
583     print("SKIPPING harvest of [movies sorted by production budget]...")
584     print()
585 else:
586     print("PERFORMING harvest of [movies sorted by production budget]...")
587     print()
588
589 COMMENTARY_LEVEL = 2
590 bgt_rank_nextup = 1
591 bgt_rank_max = 9999
592
593 comm1("====")
594 comm1(" HARVESTING DATA FROM [MULTI-PAGE TABLE]")
595 comm1(" OF MOVIES RANKED BY PRODUCTION BUDGET]")
596 comm1("====")
597 comm1("")
598
599 quite_done = False
600 issues_found = True
601 while not quite_done:
602
603     bgt_glob_path = tnURL_budgets_glob(bgt_rank_nextup, as_path=True)
604     bgt_glob_url = tnURL_budgets_glob(bgt_rank_nextup)
605
606     comm4(bgt_glob_path)
607     comm4("")
```

```
608     comm4(bgt_glob_url)
609     comm4("")  

610  

611     if not local_curated_file_exists(bgt_glob_path, bgt_glob_url):
612         print("Issue: Attempt to store webContent at URL {" +
613             bgt_glob_url + "} as local file {" +
614             bgt_glob_path + "} has failed.")
615         quite_done = True
616         break
617  

618     comm3("Getting HTML content from local file {" + bgt_glob_path + "} as soup...")
619     comm3("")  

620     try:
621         with open(bgt_glob_path, 'rb') as g:
622             localContent = g.read()
623             soup = BeautifulSoup(
624                 localContent, "html.parser")
625     except:
626         print("Issue: Attempt retrieve curated webContent from local file {" +
627             bgt_glob_path + "} has failed.")
628         quite_done = True
629         break
630  

631     comm4("Identifying tables in soup...")
632     comm4("")  

633  

634     soup_tables = soup.find_all("table")
635  

636     num_soup_tables = len(soup_tables)
637     comm4("num_soup_tables = " + repr(num_soup_tables))
638     comm4("")  

639  

640     if num_soup_tables == 0:
641         print("ISSUE: No tables were identified in soup.")
642         print("Did not obtain tabular data from file {" + bgt_glob_path + "}.")
643         quite_done = True
644         break
645  

646     if num_soup_tables > 1:
647         comm4("A total of " + str(num_soup_tables) + " tables were identified in soup.")
648         comm4("Only the first table will be examined.")
649     table = soup_tables[0]
650  

651     # Look for the header row, then extract headers from it, if found.
652     comm4("Now processing the HEADER ROW...")
653     comm4("")  

654     contains_th = lambda tag: len(tag.find_all("th")) > 0
655  

656     tr_list = [tr for tr in table.find_all("tr", recursive=False) if contains_th(tr)]
657  

658     num_header_rows = len(tr_list)
659     comm4("num_header_rows = " + repr(num_header_rows))
660     comm4("")  

661     if num_header_rows == 0:
662         print("ISSUE: No header row was identified in the table.")
663         quite_done = True
664         break
665     if num_header_rows > 1:
666         comm4("A total of " + str(num_header_rows) + " header rows were identified in the table.")
667         comm4("Only the first header row will be examined.")
668         comm4("")  

669  

670     original_headers = [".join(th.strings) for th in tr_list[0].find_all("th")]
671  

672     table_num_columns = len(original_headers)
673     comm4("table_num_columns = " + repr(table_num_columns))
674     comm4("")  

675     if table_num_columns == 0:
676         print("ISSUE: No column headers were found in the header row.")
677         quite_done = True
678         break
679  

680     comm4("The following column headers were identified:")
681     comm4(repr(original_headers))
682     comm4("")  

683
```

```

684 # correlate original headers with column indices
685 colHeader_to_j = dict(zip(original_headers, [j for j in range(table_num_columns)]))
686 j_MOVIE = colHeader_to_j['Movie']
687 j_RELEASE_DATE = colHeader_to_j['Release Date']
688 j_PRODUCTION_BUDGET = colHeader_to_j['Production Budget']
689 j_DOMESTIC_GROSS = colHeader_to_j['Domestic Gross']
690 j_WORLDWIDE_GROSS = colHeader_to_j['Worldwide Gross']
691
692 comm4("Now processing the DATA ROWS...")
693 comm4("")
694
695 contains_a = lambda tag: len(tag.find_all("a")) > 0
696 contains_td = lambda tag: len(tag.find_all("td")) > 0
697 tr_list = [tr for tr in table.find_all("tr", recursive=False) if contains_a(tr) & contains_td(tr)]
698
699 num_data_rows = len(tr_list)
700 comm4("num_data_rows = " + repr(num_data_rows))
701 comm4("")
702 if num_data_rows == 0:
703     print("ISSUE: No data rows were identified in the table.")
704     quite_done = True
705     break
706
707 for i, tr in enumerate(tr_list):
708     td_list = tr.find_all("td")
709
710     num_td = len(td_list)
711     if num_td < table_num_columns:
712         print("ISSUE: Only " + str(num_td) + " data elements were identified " +
713             "in the data row whose index is {" + str(i) + "}.")
714         print("Each data row in the table was expected to have {" + str(table_num_columns) + "} columns .")
715         quite_done = True
716         break
717
718     for tda in [td_list[j_MOVIE].a]:
719         if tda is None:
720             film_handle = None
721         else:
722             film_handle = get_movie_handle(str(tda.get('href')))
723
724     for tda in [td_list[j_RELEASE_DATE].a]:
725         found_date = None
726         if not tda is None:
727             try:
728                 found_string = tda.string
729             except:
730                 found_string = "unknown"
731
732             if found_string.lower() != "unknown":
733                 try:
734                     found_date = dateparse(found_string).date()
735                 except:
736                     print("MINOR ISSUE: unable to dateparse this: {" + repr(tda) + "}")
737
738         if found_date is None:
739             print("MINOR_ISSUE: film {" + film_handle + "} is missing a release date in {" + bgt_glob_url + "} glob.")
740
741     bgt_release_date = found_date
742
743     for td in [td_list[j_PRODUCTION_BUDGET]]:
744         if td is None:
745             bgt_productionBudget_dollars = -1
746         else:
747             bgt_productionBudget_dollars = int("".join([r for r in td.string if not r in TN_SALES OMIT]))
748
749     for td in [td_list[j_DOMESTIC_GROSS]]:
750         if td is None:
751             bgt_domUS_gross_dollars = -1
752         else:
753             bgt_domUS_gross_dollars = int("".join([r for r in td.string if not r in TN_SALES OMIT]))
754
755     for td in [td_list[j_WORLDWIDE_GROSS]]:
756         if td is None:
757             bgt_world_gross_dollars = -1
758         else:
759             bgt_world_gross_dollars = int("".join([r for r in td.string if not r in TN_SALES OMIT]))

```

```

760
761
762     bgt_doss = dict()
763     bgt_doss['01_film_handle'] = film_handle
764     bgt_doss['bgt_release_date'] = bgt_release_date
765     bgt_doss['bgt_productionBudget_dollars'] = bgt_productionBudget_dollars
766     bgt_doss['bgt_domUS_gross_dollars'] = bgt_domUS_gross_dollars
767     bgt_doss['bgt_world_gross_dollars'] = bgt_world_gross_dollars
768
769     # Check whether a dossier already exists for this movieHandle.
770     # If a dossier does not already exist, create an empty dossier for it.
771     try:
772         old_doss = movieHandle_to_movieDoss[film_handle]
773     except:
774         movieHandle_to_movieDoss[film_handle] = dict()
775
776     # Assign values for top-level keys
777     # Top-level key-value pairs contain static (non-time-dependent) information about the film
778     for k in bgt_doss.keys():
779         # Check whether key k is already defined in the dossier
780         new_val = bgt_doss[k]
781         try:
782             old_val = movieHandle_to_movieDoss[film_handle][k]
783         except:
784             movieHandle_to_movieDoss[film_handle][k] = new_val
785
786     bgt_rows_cumulative += num_data_rows
787     comm4("All {" + str(num_data_rows)+ "} data rows in the table were processed as expected.")
788     comm4("")
789
790     if num_data_rows < 100:
791         comm2("The last table has been harvested.")
792         comm2("")
793         quite_done = True
794         issues_found = False
795         break
796
797     # This tells us the rank of the first movie in the next glob to be harvested
798     bgt_rank_nextup += 100
799
800     if bgt_rank_nextup > bgt_rank_max:
801         comm2("ISSUE: The desired maximum number of rows {" + str(bgt_rank_max)+ "} has been reached.")
802         comm2("")
803         quite_done = True
804         break
805
806     if issues_found:
807         comm4("ISSUES were encountered while processing file {" + bgt_glob_path + "}.")
808         comm4("")
809     else:
810         comm2("Data from file {" + bgt_glob_path + "} was processed without issues.")
811         comm4("A total of {" + str(bgt_rows_cumulative)+ "} data rows were processed, so far.")
812         comm4("")
813
814     comm4("bgt_rows_cumulative = " + repr(bgt_rows_cumulative))
815     comm4("")
816     print("DONE PERFORMING harvest of [movies sorted by production budget].")
817     print()
818
819 harvest_tgy = True
820 if not harvest_tgy:
821     print("SKIPPING harvest of top-grossing movies by year...")
822     print()
823 else:
824     print("PERFORMING harvest of top-grossing movies by year...")
825     print()
826
827 COMMENTARY_LEVEL = 3
828
829 tgy_desired_year_first = 1977
830 tgy_desired_year_last = 2020
831 tgy_desired_years = [i for i in range(tgy_desired_year_first,
832                                         (tgy_desired_year_last + 1))]
833
834 comm1("=====")
835 comm1("    HARVESTING DATA FROM [ANNUAL TABLES OF"]

```

```

836     comm1("    OF MOVIES RANKED BY DOMESTIC GROSS")
837     comm1("    for the following box office years:")
838     comm1("    " + repr(tgy_desired_years))
839     comm1("=====")
840     comm1("")
841
842     for tgy_desired_year in tgy_desired_years:
843
844         quite_done = False
845         issues_found = True
846         while not quite_done:
847
848             comm4("=====")
849             comm4("")
850
851             comm3("Importing table of movies ranked by [domestic gross during box office year " + str(tgy_desired_year) + "]...")
852             comm4("")
853
854             tgy_table_path = tnURL_topGross_byYear(tgy_desired_year, as_path=True)
855             tgy_table_url = tnURL_topGross_byYear(tgy_desired_year)
856
857             comm4(tgy_table_path)
858             comm4("")
859             comm4(tgy_table_url)
860             comm4("")
861
862             if not local_curated_file_exists(tgy_table_path, tgy_table_url):
863                 print("Issue: Attempt to store webContent at URL {" +
864                     tgy_table_url + "} as local file {" +
865                     tgy_table_path + "} has failed.")
866                 quite_done = True
867                 break
868
869             comm4("Getting HTML content from local file {" + tgy_table_path + "} as soup...")
870             comm4("")
871             try:
872                 with open(tgy_table_path, 'rb') as g:
873                     localContent = g.read()
874                     soup = BeautifulSoup(
875                         localContent, "html.parser")
876             except:
877                 print("Issue: Attempt retrieve curated webContent from local file {" +
878                     tgy_table_path + "} has failed.")
879                 quite_done = True
880                 break
881
882             comm4("Identifying tables in soup...")
883             comm4("")
884
885             soup_tables = soup.find_all("table")
886
887             num_soup_tables = len(soup_tables)
888             comm4("num_soup_tables = " + repr(num_soup_tables))
889             comm4("")
890
891             if num_soup_tables == 0:
892                 print("ISSUE: No tables were identified in soup.")
893                 print("Did not obtain tabular data from URL {" + tgy_table_url + "}.")
894                 quite_done = True
895                 break
896
897             if num_soup_tables > 1:
898                 comm4("A total of " + str(num_soup_tables) + " tables were identified in soup.")
899                 comm4("Only the first table will be examined.")
900             table = soup_tables[0]
901
902             # Look for the header row, then extract headers from it, if found.
903             comm4("Now processing the HEADER ROW...")
904             comm4("")
905             contains_th = lambda tag: len(tag.find_all("th")) > 0
906
907             tr_list = [tr for tr in table.find_all("tr", recursive=False) if contains_th(tr)]
908
909             num_header_rows = len(tr_list)
910             comm4("num_header_rows = " + repr(num_header_rows))
911             comm4("")

```

```

912     if num_header_rows == 0:
913         print("ISSUE: No header row was identified in the table.")
914         quite_done = True
915         break
916     if num_header_rows > 1:
917         comm4("A total of " + str(num_header_rows) + " header rows were identified in the table.")
918         comm4("Only the first header row will be examined.")
919         comm4("")
920
921     original_headers = [" ".join(th.strings) for th in tr_list[0].find_all("th")]
922
923     table_num_columns = len(original_headers)
924     comm4("table_num_columns = " + repr(table_num_columns))
925     comm4("")
926     if table_num_columns == 0:
927         print("ISSUE: No column headers were found in the header row.")
928         quite_done = True
929         break
930
931     comm4("The following column headers were identified:")
932     comm4(repr(original_headers))
933     comm4("")
934
935     # correlate original headers with column indices
936     colHeader_to_j = dict(zip(original_headers, [j for j in range(table_num_columns)]))
937     j_YYYY_DOMUS_GROSS_RANK = colHeader_to_j['Rank']
938     j_MOVIE = colHeader_to_j['Movie']
939     j_RELEASE_DATE = colHeader_to_j['Release Date']
940     j_DISTRIBUTOR = colHeader_to_j['Distributor']
941     j_GENRE = colHeader_to_j['Genre']
942     j_YYYY_DOMUS_GROSS = colHeader_to_j[str(tgy_desired_year) + ' Gross']
943     j_YYYY_DOMUS_TICKETS = colHeader_to_j['Tickets Sold']
944
945     table_BOX_OFFICE_YEAR = original_headers[j_YYYY_DOMUS_GROSS][0:4]
946     comm4("table_BOX_OFFICE_YEAR = " + table_BOX_OFFICE_YEAR)
947     comm4("")
948
949     # Make sure the box office year embedded in the column header is the
950     # same as the desired year
951     if table_BOX_OFFICE_YEAR != str(tgy_desired_year):
952         print("ISSUE: A column header mentions year {" + table_BOX_OFFICE_YEAR +
953             "}, whereas {" + str(tgy_desired_year) + "} is the desired year.")
954         quite_done = True
955         break
956
957     comm4("Now processing the DATA ROWS...")
958     comm4("")
959
960     contains_a = lambda tag: len(tag.find_all("a")) > 0
961     contains_td = lambda tag: len(tag.find_all("td")) > 0
962     tr_list = [tr for tr in table.find_all("tr", recursive=False) if contains_a(tr) & contains_td(tr)]
963
964     num_data_rows = len(tr_list)
965     comm4("num_data_rows = " + repr(num_data_rows))
966     comm4("")
967     if num_data_rows == 0:
968         print("ISSUE: No data rows were identified in the table.")
969         quite_done = True
970         break
971
972     for i, tr in enumerate(tr_list):
973         td_list = tr.find_all("td")
974
975         num_td = len(td_list)
976         if num_td < table_num_columns:
977             print("ISSUE: Only " + str(num_td) + " data elements were identified " +
978                 "in the data row whose index is {" + str(i) + "}.")
979             print("Each data row in the table was expected to have {" + str(table_num_columns) + "} columns .")
980             quite_done = True
981             break
982
983         tgy_YYYY = table_BOX_OFFICE_YEAR
984
985         for td in [td_list[j_YYYY_DOMUS_GROSS_RANK]]:
986             if td is None:
987                 tgy_domUssales_dollars_rank_boxOfficeYear = None

```

```

988
989         tgy_domUssales_dollars_rank_boxOfficeYear = int(td.string)
990
991     for tda in [td_list[j_MOVIE].a]:
992         if tda is None:
993             film_handle = None
994             tgy_film_href = None
995             tgy_film_title = None
996             tgy_film_url = None
997         else:
998             film_handle = get_movie_handle(str(tda.get('href')))
999             tgy_film_href = str(tda.get('href'))
1000             tgy_film_title = tda.string
1001             tgy_film_url = TN_MAIN_URL + str(tda.get('href'))
1002
1003     for tda in [td_list[j_RELEASE_DATE].a]:
1004
1005         found_date = None
1006         if not tda is None:
1007             try:
1008                 found_string = tda.string
1009             except:
1010                 found_string = "unknown"
1011
1012             if found_string.lower() != "unknown":
1013                 try:
1014                     found_date = dateparse(found_string).date()
1015                 except:
1016                     print("MINOR_ISSUE: unable to dateparse this: {" + repr(tda) + "}.")
1017
1018         if found_date is None:
1019             comm2("MINOR_ISSUE: film {" + film_handle +
1020                   "} is missing a release date in the {" +
1021                   tgy_YYYY + "_topGross} table.")
1022
1023     tgy_release_date = found_date
1024
1025     if tda is None:
1026         tgy_release_href = None
1027         tgy_release_url = None
1028     else:
1029         tgy_release_href = str(tda.get('href'))
1030         tgy_release_url = TN_MAIN_URL + str(tda.get('href'))
1031
1032     for tda in [td_list[j_DISTRIBUTOR].a]:
1033         if tda is None:
1034             tgy_distrib_href = None
1035             tgy_distrib_handle = None
1036             tgy_distrib_name = None
1037             tgy_distrib_url = None
1038         else:
1039             tgy_distrib_href = str(tda.get('href'))
1040             tgy_distrib_handle = get_distrib_handle(str(tda.get('href')))
1041             tgy_distrib_name = tda.string
1042             tgy_distrib_url = TN_MAIN_URL + str(tda.get('href'))
1043
1044     for tda in [td_list[j_GENRE].a]:
1045         if tda is None:
1046             tgy_genre_href = None
1047             tgy_genre_handle = TN_GENRE_HANDLES[0]
1048             tgy_genre_abbr = TN_GENRE_ABBRS[0]
1049             tgy_genre_name = None
1050             tgy_genre_url = None
1051         else:
1052             tgy_genre_href = str(tda.get('href'))
1053             tgy_genre_handle = get_genre_handle(str(tda.get('href')))
1054             tgy_genre_abbr = gHandle_to_gAbbr[get_genre_handle(str(tda.get('href')))]
1055             tgy_genre_name = tda.string
1056             tgy_genre_url = TN_MAIN_URL + str(tda.get('href'))
1057
1058     for td in [td_list[j_YYYY_DOMUS_GROSS]]:
1059         if td is None:
1060             tgy_domUssales_dollars_boxOfficeYear = -1
1061         else:
1062             tgy_domUssales_dollars_boxOfficeYear = int("".join([r for r in td.string if not r in TN_SALES OMIT]))
1063

```

```

1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139

    for td in [td_list[j_YYYY_DOMUS_TICKETS]]:
        if td is None:
            tgy_domUssales_tickets_boxOfficeYear = -1
        else:
            tgy_domUssales_tickets_boxOfficeYear = int("".join([r for r in td.string if not r in TN_SEATS OMIT]))

    if COMMENTARY_LEVEL >= 4:
        print("01_film_handle = " + repr(film_handle))
        print("tgy_film_title = " + repr(tgy_film_title))
        print("tgy_film_url = " + repr(tgy_film_url))
        print("tgy_release_date = " + repr(tgy_release_date))
        print("tgy_release_url = " + repr(tgy_release_url))
        print("tgy_distrib_handle = " + repr(tgy_distrib_handle))
        print("tgy_distrib_name = " + repr(tgy_distrib_name))
        print("tgy_distrib_url = " + repr(tgy_distrib_url))
        print("tgy_genre_handle = " + repr(tgy_genre_handle))
        print("tgy_genre_abbr = " + repr(tgy_genre_abbr))
        print("tgy_genre_name = " + repr(tgy_genre_name))
        print("tgy_genre_url = " + repr(tgy_genre_url))
        print("tgy_YYYY = " + repr(tgy_YYYY))
        print("tgy_domUssales_dollars_rank" + tgy_YYYY + " = " + repr(tgy_domUssales_dollars_rank_boxOfficeYear))
        print("tgy_domUssales_dollars_" + tgy_YYYY + " = " + repr(tgy_domUssales_dollars_boxOfficeYear))
        print("tgy_domUssales_tickets_" + tgy_YYYY + " = " + repr(tgy_domUssales_tickets_boxOfficeYear))
        print("")
        print("====")
        print("")

    tgy_doss = dict()
    tgy_doss['01_film_handle'] = film_handle
    tgy_doss['03_genre_abbr'] = tgy_genre_abbr
    # tgy_doss['tgy_film_title'] = tgy_film_title
    # tgy_doss['tgy_film_url'] = tgy_film_url
    tgy_doss['04_distrib_handle'] = tgy_distrib_handle
    tgy_doss['tgy_boxOfficeYears'] = tgy_desired_year
    tgy_doss['tgy_release_dates'] = tgy_release_date
    # tgy_doss['tgy_domUssales_dollars_rank' + tgy_YYYY] = tgy_domUssales_dollars_rank_boxOfficeYear
    tgy_doss['tgy_domUssales_dollars_' + tgy_YYYY] = tgy_domUssales_dollars_boxOfficeYear
    tgy_doss['tgy_domUssales_tickets_' + tgy_YYYY] = tgy_domUssales_tickets_boxOfficeYear

    # Check whether a dossier already exists for this movieHandle.
    # If a dossier does not already exist, create an empty dossier for it.
    try:
        old_doss = movieHandle_to_movieDoss[film_handle]
    except:
        movieHandle_to_movieDoss[film_handle] = dict()
    # Assign values for top-level keys
    # Top-level key-value pairs contain static (non-time-dependent) information about the film
    for k in tgy_doss.keys():
        new_val = tgy_doss[k]
        if k in ["tgy_boxOfficeYears", "tgy_release_dates"]:
            # Check whether this k-list is already present for this movieHandle
            # If the k-list does not already exist, create it as an empty list.
            try:
                old_val = movieHandle_to_movieDoss[film_handle][k]
            except:
                movieHandle_to_movieDoss[film_handle][k] = list()
            # Append the new value to the k-list
            movieHandle_to_movieDoss[film_handle][k].append(new_val)

        if k in ["tgy_release_dates"]:
            # force update of "tgy_release_date", because a new date
            # entry may exist for the current movieHandle
            dates_raw = movieHandle_to_movieDoss[film_handle][k]
            dates_valid = [d for d in dates_raw if isinstance(d, datetime.date)]
            if len(dates_valid) == 0:
                date_earliest = None
            else:
                date_earliest = min(dates_valid)
                if not isinstance(date_earliest, datetime.date):
                    print("The earliest date in dates_valid is not a date..." + repr(dates_valid))
                    print("dates_valid = " + repr(dates_valid))
                    print("date_earliest = " + repr(date_earliest))
                    print()
            movieHandle_to_movieDoss[film_handle]["tgy_release_date"] = date_earliest
        else:
            pass

```

```

1140         # Check whether key k is already defined in the dossier
1141         try:
1142             old_val = movieHandle_to_movieDoss[film_handle][k]
1143         except:
1144             movieHandle_to_movieDoss[film_handle][k] = new_val
1145
1146     if not tgy_distrib_handle is None:
1147         distrib_doss = dict()
1148         distrib_doss['distrib_href'] = tgy_distrib_href
1149         distrib_doss['distrib_handle'] = tgy_distrib_handle
1150         distrib_doss['distrib_name'] = tgy_distrib_name
1151         distrib_doss['distrib_url'] = tgy_distrib_url
1152
1153         # Check whether a dossier already exists for this distribHandle.
1154         # If a dossier does not already exist, create an empty dossier for it.
1155         try:
1156             old_doss = distribHandle_to_distribDoss[tgy_distrib_handle]
1157         except:
1158             distribHandle_to_distribDoss[tgy_distrib_handle] = dict()
1159
1160         # Assign values for top-level keys
1161         # Top-Level key-value pairs contain static (non-time-dependent) information about the film
1162         for k in distrib_doss.keys():
1163             # Check whether key k is already defined in the dossier
1164             new_val = distrib_doss[k]
1165             try:
1166                 old_val = distribHandle_to_distribDoss[tgy_distrib_handle][k]
1167             except:
1168                 distribHandle_to_distribDoss[tgy_distrib_handle][k] = new_val
1169
1170     if not tgy_genre_handle is None:
1171         genre_doss = dict()
1172         genre_doss['genre_href'] = tgy_genre_href
1173         genre_doss['genre_handle'] = tgy_genre_handle
1174         genre_doss['genre_abbr'] = tgy_genre_abbr
1175         genre_doss['genre_name'] = tgy_genre_name
1176         genre_doss['genre_url'] = tgy_genre_url
1177
1178         # Check whether a dossier already exists for this genreHandle.
1179         # If a dossier does not already exist, create an empty dossier for it.
1180         try:
1181             old_doss = genreHandle_to_genreDoss[tgy_genre_handle]
1182         except:
1183             genreHandle_to_genreDoss[tgy_genre_handle] = dict()
1184
1185         # Assign values for top-level keys
1186         # Top-Level key-value pairs contain static (non-time-dependent) information about the film
1187         for k in genre_doss.keys():
1188             # Check whether key k is already defined in the dossier
1189             new_val = genre_doss[k]
1190             try:
1191                 old_val = genreHandle_to_genreDoss[tgy_genre_handle][k]
1192             except:
1193                 genreHandle_to_genreDoss[tgy_genre_handle][k] = new_val
1194
1195         tgy_rows_cumulative += num_data_rows
1196         comm4("All {" + str(num_data_rows)+ " } data rows in the table were processed as expected.")
1197         comm4("")
1198
1199         quite_done = True
1200         issues_found = False
1201
1202     if issues_found:
1203         print("ISSUES were encountered while processing URL {" + tgy_table_url + "}.")
1204         comm2("A total of {" + str(tgy_rows_cumulative)+ " } data rows were processed, so far.")
1205         comm2("")
1206     else:
1207         comm2("Data from URL {" + tgy_table_url + " } was processed without issues.")
1208         comm4("A total of {" + str(tgy_rows_cumulative)+ " } data rows were processed, so far.")
1209         comm4("")
1210
1211 # =====
1212 # IN THIS SECTION,
1213 # POST-PROCESSING OF THE DATA HARVESTED ABOVE IS PERFORMED.
1214 #
1215 # EACH MODE OF POST-HARVEST PROCESSING
1216 # MAY OPTIONALLY BE PERFORMED AT THE DISCRETION OF THE USER

```

```

1216 #      BY SETTING "identify_????" FLAGS AND/OR
1217 #      "mend_????" FLAGS
1218 #      TO True or False, ACCORDINGLY
1219 # =====
1220
1221
1222 identify_unique_movieHandles = True
1223 if not identify_unique_movieHandles:
1224     print("SKIPPING identification of unique movie handles...")
1225     print()
1226 else:
1227     print("IDENTIFYING unique movie handles...")
1228     print()
1229
1230     # Make a list of all unique movie handles that were encountered
1231     # during the harvesting of data
1232     movieHandles_all = sorted([k for k in movieHandle_to_movieDoss.keys()],
1233                             key=lambda s: s.lower())
1234     num_movieHandles = len(movieHandles_all)
1235     print("num_movieHandles = " + str(num_movieHandles))
1236     print()
1237     num_display = 50
1238     print("MOVIE HANDLES: FIRST " + str(num_display) +
1239           " AND LAST " + str(num_display) + "...")
1240     i_all = list(range(num_movieHandles))
1241     i_show = i_all[0:num_display] + i_all[-num_display:]
1242     for i in i_show:
1243         print("movieHandles_all[" + str(i) + "] = " + movieHandles_all[i])
1244     print()
1245
1246 identify_unique_genreHandles = True
1247 if not identify_unique_genreHandles:
1248     print("SKIPPING identification of unique genre handles...")
1249     print()
1250 else:
1251     print("IDENTIFYING unique genre handles...")
1252     print()
1253
1254     # Make a list of all unique genre handles that were encountered
1255     # during the harvesting of data
1256     genreHandles_all = sorted([k for k in genreHandle_to_genreDoss.keys()],
1257                               key=lambda s: s.lower())
1258     num_genreHandles = len(genreHandles_all)
1259     print("num_genreHandles = " + str(num_genreHandles))
1260     print()
1261     num_display = 50
1262     if num_genreHandles < (2 * num_display):
1263         print("GENRE HANDLES: ALL " + str(num_genreHandles) + " HANDLES...")
1264         i_show = list(range(num_genreHandles))
1265     else:
1266         print("GENRE HANDLES: FIRST " + str(num_display) +
1267               " AND LAST " + str(num_display) + "...")
1268         i_all = list(range(num_genreHandles))
1269         i_show = i_all[0:num_display] + i_all[-num_display:]
1270     for i in i_show:
1271         print("genreHandles_all[" + str(i) + "] = " + genreHandles_all[i])
1272     print()
1273
1274 identify_unique_distribHandles = True
1275 if not identify_unique_distribHandles:
1276     print("SKIPPING identification of unique distributor handles...")
1277     print()
1278 else:
1279     print("IDENTIFYING unique distributor handles...")
1280     print()
1281
1282     # Make a list of all unique distributor handles that were encountered
1283     # during the harvesting of data
1284     distribHandles_all = sorted([k for k in distribHandle_to_distribDoss.keys()],
1285                                key=lambda s: s.lower())
1286     num_distribHandles = len(distribHandles_all)
1287     print("num_distribHandles = " + str(num_distribHandles))
1288     print()
1289     num_display = 50
1290     if num_distribHandles < (2 * num_display):
1291         print("DISTRIBUTOR HANDLES: ALL " + str(num_distribHandles) + " HANDLES...")

```

```

1292     i_show = list(range(num_distribHandles))
1293 else:
1294     print("DISTRIBUTOR HANDLES: FIRST " + str(num_display) +
1295          " AND LAST " + str(num_display) + "...")
1296     i_all = list(range(num_distribHandles))
1297     i_show = i_all[:num_display] + i_all[-num_display:]
1298 for i in i_show:
1299     print("distribHandles_all[" + str(i) + "] = " + distribHandles_all[i])
1300 print()
1301
1302 mend_genre_abbrs = True
1303 if not mend_genre_abbrs:
1304     print("SKIPPING mending of genre abbreviations...")
1305     print()
1306 else:
1307     print("PERFORMING mending of genre abbreviations...")
1308     print()
1309
1310 # Not all movies that were encountered during the harvesting of data
1311 # had genre information available to be stored in the movie's dossier.
1312
1313
1314 # Force those movies that are MISSING the genre information to have
1315 # genre abbreviation TN_GENRE_ABBRS[0], which is regarded as "undefined genre"
1316
1317 for mh in movieHandles_all:
1318     g_abbr = None
1319     try:
1320         g_abbr = movieHandle_to_movieDoss[mh]["03_genre_abbr"]
1321     except:
1322         pass
1323
1324     if not g_abbr in TN_GENRE_ABBRS:
1325         movieHandle_to_movieDoss[mh]["03_genre_abbr"] = TN_GENRE_ABBRS[0]
1326
1327
1328 mend_release_dates = True
1329 if not mend_release_dates:
1330     print("SKIPPING mending of release dates...")
1331     print()
1332 else:
1333     print("PERFORMING mending of release dates...")
1334     print()
1335
1336 # Not all movies that were encountered during the harvesting of data
1337 # had "release date" information available to be stored in the movie's dossier.
1338
1339 # To complicate things further, the "release date" information harvested from
1340 # source A (if release date information was in fact obtained from source A)
1341 # MAY differ from the "release date" information harvested from source B (if
1342 # release date information was in fact obtained from source B).
1343
1344 # In the event that there appear to be two different values of "release date"
1345 # for the same movie, the EARLIER of the two values will be regarded as correct.
1346
1347 for mh in movieHandles_all:
1348     date_A = "--absent--"
1349     try:
1350         date_A = movieHandle_to_movieDoss[mh]["bgt_release_date"]
1351     except:
1352         pass
1353
1354     date_B = "--absent--"
1355     try:
1356         date_B = movieHandle_to_movieDoss[mh]["tgy_release_date"]
1357     except:
1358         pass
1359
1360     if not isinstance(date_A, datetime.date):
1361         if not isinstance(date_B, datetime.date):
1362             date_release = None
1363         else:
1364             date_release = date_B
1365     else:
1366         if not isinstance(date_B, datetime.date):
1367             date_release = date_A

```

```

1368     else:
1369         date_release = min(date_A, date_B)
1370
1371     movieHandle_to_movieDoss[mh]["02_film_release_date"] = date_release
1372     year_release = date_to_boxOfficeYear(date_release)
1373     if isinstance(year_release, int):
1374         pass
1375     if not isinstance(year_release, int):
1376         if not year_release is None:
1377             print("Error: date_to_boxOfficeYear(" + repr(date_release)+ ") = " + repr(year_release))
1378         movieHandle_to_movieDoss[mh]["02_film_release_year"] = year_release
1379
1380     print("all movieHandles whose 'release_date' is a non-date, AFTER mending of release dates:")
1381     movieHandles_undated = [mh for mh in movieHandles_all if not
1382                             isinstance(movieHandle_to_movieDoss[mh]['02_film_release_date'], datetime.date)]
1383     for mh in movieHandles_undated:
1384         mh_date_keys = [k for k in movieHandle_to_movieDoss[mh].keys() if "date" in k]
1385         mh_date_values = [movieHandle_to_movieDoss[mh][dk] for dk in mh_date_keys]
1386         mh_date = dict(zip(mh_date_keys, mh_date_values))
1387         print(" " + repr(mh) + ": " + repr(mh_date) )
1388     print()
1389
1390
1391 display_selected_movie_dossiers = True
1392 if not display_selected_movie_dossiers:
1393     print("SKIPPING display of selected movie dossiers...")
1394     print()
1395 else:
1396     print("DISPLAYING selected movie dossiers...")
1397     print()
1398
1399 num_display = 20
1400 print("MOVIE DOSSIERS: FIRST " + str(num_display) +
1401      " AND LAST " + str(num_display) + "...")
1402 i_all = list(range(num_movieHandles))
1403 i_show = i_all[0:num_display] + i_all[-num_display:]
1404 for i in i_show:
1405     print("i = " + str(i) + ":")
1406     mh_doss = movieHandle_to_movieDoss[movieHandles_all[i]]
1407     mh_keys = sorted(mh_doss.keys(), key=lambda s: s.lower())
1408     for k in mh_keys:
1409         print(repr(k) + ": " + repr(mh_doss[k]))
1410     print()
1411
1412
1413
1414 # =====
1415 # IN THIS SECTION,
1416 #   SOME OF THE INFORMATION STORED IN movieHandle_to_movieDoss
1417 #   IS PORTED INTO A NEW DICTIONARY CALLED "eventHandle_to_eventDoss"
1418 #
1419 #   TO BE PORTED INTO THE EVENTS DICTIONARY, A PARTICULAR MOVIE
1420 #   MUST ALREADY CONTAIN SPECIFIC ITEMS OF INFORMATION IN ITS MOVIE DOSSIER.
1421 #
1422 #   EACH MOVIE THAT APPEARS IN THE EVENTS DICTIONARY HAS AT LEAST ONE
1423 #   "EVENT" ASSOCIATED WITH IT.
1424 #
1425 #   THE COMBINATION OF:
1426 #       [ONE BOX-OFFICE SALES YEAR] + [ONE GENRE] + [ONE MOVIE FROM WITHIN THAT GENRE]
1427 #   COMPRIZE A UNIQUE "EVENT HANDLE".
1428 #
1429 #   FOR EXAMPLE, THE 2009 ACTION MOVIE "AVATAR" HAD DOMESTIC US BOX OFFICE
1430 #   SALES DURING THE BOX OFFICE YEARS 2009 AND 2010. THE INFORMATION FOR
1431 #   THE MOVIE AVATAR IS THEREFORE STORED IN THE EVENTS DICTIONARY
1432 #   UNDER THESE TWO EVENT HANDLES.
1433 #       EVENT HANDLE 1 = "2009/ACTION/AVATAR"
1434 #       EVENT HANDLE 2 = "2010/ACTION/AVATAR"
1435 #
1436 #   ORGANIZATION OF THE MOVIE SALES FIGURES IN THIS FASHION FACILITATES
1437 #   ANALYSIS USING THE PANDAS "GROUPBY" FUNCTION.
1438 # =====
1439
1440
1441 movieHandles_in_eventDoss = []
1442 create_eventHandle_to_eventDoss = True

```

```

1444 if not create_eventHandle_to_eventDoss:
1445     print("SKIPPING creation of eventHandle_to_eventDoss...")
1446     print()
1447 else:
1448     print("PERFORMING creation of eventHandle_to_eventDoss...")
1449     print()
1450
1451 eventHandle_to_eventDoss = dict()
1452
1453 for mh in movieHandles_all:
1454     mh_BOYs = None
1455     try:
1456         mh_BOYs = movieHandle_to_movieDoss[mh]['tgy_boxOfficeYears']
1457     except:
1458         print("didn't find movieHandle_to_movieDoss[" + repr(mh) + "]['tgy_boxOfficeYears']")
1459         pass
1460     if isinstance(mh_BOYs, list):
1461         mh_relDate = None
1462         try:
1463             mh_relDate = movieHandle_to_movieDoss[mh]['02_film_release_date']
1464         except:
1465             print("didn't find movieHandle_to_movieDoss[" + repr(mh) + "]['02_film_release_date']")
1466             pass
1467     if isinstance(mh_relDate, datetime.date):
1468         mh_pb = None
1469         try:
1470             mh_pb = movieHandle_to_movieDoss[mh]['bgt_productionBudget_dollars']
1471         except:
1472             print("didn't find movieHandle_to_movieDoss[" + repr(mh) + "]['bgt_productionBudget_dollars']")
1473             pass
1474     if isinstance(mh_pb, int):
1475         mh_domUS_gross = None
1476         try:
1477             mh_domUS_gross = movieHandle_to_movieDoss[mh]['bgt_domUS_gross_dollars']
1478         except:
1479             print("didn't find movieHandle_to_movieDoss[" + repr(mh) + "]['bgt_domUS_gross_dollars']")
1480             pass
1481     if isinstance(mh_domUS_gross, int):
1482         mh_world_gross = None
1483         try:
1484             mh_world_gross = movieHandle_to_movieDoss[mh]['bgt_world_gross_dollars']
1485         except:
1486             print("didn't find movieHandle_to_movieDoss[" + repr(mh) + "]['bgt_world_gross_dollars']")
1487             pass
1488     if isinstance(mh_world_gross, int):
1489         for iYear in mh_BOYs:
1490             event_iYear = iYear
1491             event_YYYY = str(event_iYear)
1492             event_genre_abbr = movieHandle_to_movieDoss[mh]['03_genre_abbr']
1493             event_movie_handle = mh
1494
1495             event_handle = event_YYYY + "/" + event_genre_abbr + "/" + event_movie_handle
1496             event_genreYear_handle = event_genre_abbr + "/" + event_YYYY
1497
1498             mh_sales = None
1499             try:
1500                 mh_sales = movieHandle_to_movieDoss[mh]['tgy_domUSSales_dollars_' + event_YYYY]
1501             except:
1502                 pass
1503             if isinstance(mh_sales, int):
1504                 mh_tickets = None
1505                 try:
1506                     mh_tickets = movieHandle_to_movieDoss[mh]['tgy_domUSSales_tickets_' + event_YYYY]
1507                 except:
1508                     pass
1509             if isinstance(mh_tickets, int):
1510                 pb_year = movieHandle_to_movieDoss[mh]['02_film_release_year']
1511                 pb_raw = movieHandle_to_movieDoss[mh]['bgt_productionBudget_dollars']
1512                 pb_mult = inflation_multiplier(from_year=pb_year, into_year=2020)
1513                 pb_adj = pb_raw * pb_mult
1514
1515                 domUS_raw = movieHandle_to_movieDoss[mh]['tgy_domUSSales_dollars_' + event_YYYY]
1516                 domUS_mult = inflation_multiplier(from_year=event_iYear, into_year=2020)
1517                 domUS_adj = domUS_raw * domUS_mult
1518
1519             event_doss = dict()

```

```

1520     event_doss["event_handle"] = event_handle
1521     event_doss["genreYear_handle"] = event_genreYear_handle
1522     event_doss['boxOfficeYear'] = event_iYear
1523     event_doss["genre_abbr"] = event_genre_abbr
1524     event_doss["movie_handle"] = event_movie_handle
1525     event_doss["release_date"] = movieHandle_to_movieDoss[mh]['02_film_release_date']
1526     event_doss["productionBudget_year"] = pb_year
1527     event_doss["productionBudget_raw"] = pb_raw
1528     event_doss["productionBudget_mult"] = pb_mult
1529     event_doss["productionBudget_adj"] = pb_adj
1530     event_doss["total_world_gross"] = movieHandle_to_movieDoss[mh]['bgt_world_gross_dollars']
1531     event_doss["total_domUS_gross"] = movieHandle_to_movieDoss[mh]['bgt_domUS_gross_dollars']
1532     event_doss["boxOfficeYear_domUS_raw"] = domUS_raw
1533     event_doss["boxOfficeYear_domUS_mult"] = domUS_mult
1534     event_doss["boxOfficeYear_domUS_adj"] = domUS_adj
1535     event_doss["boxOfficeYear_domUS_tickets"] = movieHandle_to_movieDoss[mh]['tgy_domUSSales_tickets_' + event_YYYY]
1536     eventHandle_to_eventDoss[event_handle] = event_doss
1537
1538     movieHandles_in_eventDoss.append(mh)
1539 print()
1540
1541 num_movieHandles_in_eventDoss = len(movieHandles_in_eventDoss)
1542 print("BEFORE ELIMINATION OF DUPLICATES:")
1543 print("num_movieHandles_in_eventDoss = " + str(num_movieHandles_in_eventDoss))
1544 print()
1545
1546 movieHandles_in_eventDoss = sorted(list(set(movieHandles_in_eventDoss)))
1547 num_movieHandles_in_eventDoss = len(movieHandles_in_eventDoss)
1548 print("AFTER ELIMINATION OF DUPLICATES:")
1549 print("num_movieHandles_in_eventDoss = " + str(num_movieHandles_in_eventDoss))
1550 print()
1551
1552
1553 eventHandles_all = sorted([k for k in eventHandle_to_eventDoss.keys()],
1554                           key=lambda s: s.lower())
1555 num_eventHandles = len(eventHandles_all)
1556 print("num_eventHandles = " + str(num_eventHandles))
1557 print()
1558
1559 # make the event handles in the events dictionary appear in increasing order by eventHandle
1560 eventHandle_to_eventDoss_RAW = eventHandle_to_eventDoss
1561 eventHandle_to_eventDoss = dict()
1562 for k in eventHandles_all:
1563     eventHandle_to_eventDoss[k] = eventHandle_to_eventDoss_RAW[k]
1564 eventHandle_to_eventDoss_RAW = None
1565
1566 num_display = 50
1567 print("EVENT HANDLES: FIRST " + str(num_display) +
1568       " AND LAST " + str(num_display) + "...")
1569 i_all = list(range(num_eventHandles))
1570 i_show = i_all[0:num_display] + i_all[-num_display:]
1571 for i in i_show:
1572     print("eventHandles_all[" + str(i) + "] = " + eventHandles_all[i])
1573 print()
1574
1575 num_display = 20
1576 print("EVENT DOSSIERS: FIRST " + str(num_display) +
1577       " AND LAST " + str(num_display) + "...")
1578 i_all = list(range(num_eventHandles))
1579 i_show = i_all[0:num_display] + i_all[-num_display:]
1580 for i in i_show:
1581     print("i = " + str(i) + ":")
1582     eh_doss = eventHandle_to_eventDoss[eventHandles_all[i]]
1583     eh_keys = eh_doss.keys()
1584     for k in eh_keys:
1585         print(repr(k) + ": " + repr(eh_doss[k]))
1586     print()
1587
1588
1589 # Make a sound, to let the user know it's time to move on to the next cell
1590 make_beeps()
1591
1592 # That should do it. Thanks for reading my code. -- MLC
1593 print("Get thee to the next cell.")
1594
1595 do_extra_stuff = False

```

```

1596 if do_extra_stuff:
1597     # This is extra stuff
1598     # so please don't knock me for lack of commentary
1599
1600     further_review_data = False
1601     if not further_review_data:
1602         print("SKIPPING further review of data...")
1603         print()
1604     else:
1605         print("PERFORMING further review of data...")
1606         print()
1607
1608         print("bgt_rows_cumulative = " + str(bgt_rows_cumulative))
1609         print()
1610
1611         print("tgy_rows_cumulative = " + str(tgy_rows_cumulative))
1612         print()
1613
1614         movieHandles_length = [len(h) for h in movieHandles_all]
1615         movieHandles_length_min = min(movieHandles_length)
1616         movieHandles_length_max = max(movieHandles_length)
1617         movieHandles_shortest = [h for h in movieHandles_all if len(h) == movieHandles_length_min]
1618         movieHandles_longest = [h for h in movieHandles_all if len(h) == movieHandles_length_max]
1619         print("movieHandles_length_min = " + str(movieHandles_length_min))
1620         print("movieHandles_length_max = " + str(movieHandles_length_max))
1621         print("movieHandles_shortest = " + repr(movieHandles_shortest))
1622         print("movieHandles_longest = " + repr(movieHandles_longest))
1623         print()
1624
1625     try:
1626         movieHandles_with_releaseDate = [h for h in movieHandles_all if not movieHandle_to_movieDoss[h]['02_film_release_date'] is None]
1627         num_movieHandles_with_releaseDate = len(movieHandles_with_releaseDate)
1628         print("num_movieHandles_with_releaseDate = " + str(num_movieHandles_with_releaseDate))
1629         print()
1630
1631
1632         movieHandles_releaseDate = [movieHandle_to_movieDoss[h]['02_film_release_date'] for h in movieHandles_with_releaseDate]
1633         movieHandles_releaseDate_min = min(movieHandles_releaseDate)
1634         movieHandles_releaseDate_max = max(movieHandles_releaseDate)
1635         movieHandles_earliest = [h for h in movieHandles_with_releaseDate if
1636             movieHandle_to_movieDoss[h]['02_film_release_date'] == movieHandles_releaseDate_min]
1637         movieHandles_latest = [h for h in movieHandles_with_releaseDate if
1638             movieHandle_to_movieDoss[h]['02_film_release_date'] == movieHandles_releaseDate_max]
1639         print("movieHandles_releaseDate_min = " + str(movieHandles_releaseDate_min))
1640         print("movieHandles_releaseDate_max = " + str(movieHandles_releaseDate_max))
1641         print("movieHandles_earliest = " + repr(movieHandles_earliest))
1642         print("movieHandles_latest = " + repr(movieHandles_latest))
1643         print()
1644     except:
1645         print("skipping movieHandles_releaseDate analysis")
1646         print()
1647
1648     try:
1649         movieHandles_numBOY = [len(movieHandle_to_movieDoss[h]['boxOfficeYears']) for h in movieHandles_all]
1650         movieHandles_numBOY_min = min(movieHandles_numBOY)
1651         movieHandles_numBOY_max = max(movieHandles_numBOY)
1652         movieHandles_numBOY_highest = [for h, numBOY in zip(movieHandles_all, movieHandles_numBOY)
1653             if numBOY == movieHandles_numBOY_max]
1654         print("movieHandles_numBOY_min = " + str(movieHandles_numBOY_min))
1655         print("movieHandles_numBOY_max = " + str(movieHandles_numBOY_max))
1656         print("movieHandles_numBOY_highest = " + str(movieHandles_numBOY_highest))
1657         for mh in movieHandles_numBOY_highest:
1658             print("movieHandle_to_movieDoss[" + repr(mh) + "]['boxOfficeYears'] = ")
1659             print(repr(movieHandle_to_movieDoss[mh]['boxOfficeYears']))
1660             print()
1661     except:
1662         print("skipping movieHandles_numBOY analysis")
1663         print()
1664
1665     # This is a nice way to end program execution, in a Jupyter Notebook
1666     print("Purposely Terminating program execution, now.")
1667     assert False
1668

```

```
'productionBudget_year': 2019
'productionBudget_raw': 65000000
'productionBudget_mult': 1.00934
'productionBudget_adj': 65607099.9999999
'total_world_gross': 39194264
'total_domUS_gross': 17291078
'boxOfficeYear_domUS_raw': 17291078
'boxOfficeYear_domUS_mult': 1.0
'boxOfficeYear_domUS_adj': 17291078.0
'boxOfficeYear_domUS_tickets': 1898032

i = 6785:
'event_handle': '2020/THRILLER/Unhinged-(2020)'
'genreYear_handle': 'THRILLER/2020'
'boxOfficeYear': 2020
'genre_abbr': 'THRILLER'
'movie_handle': 'Unhinged-(2020)'
'release_date': datetime.date(2020, 7, 16)
'productionBudget_year': 2020
'productionBudget_raw': 33000000
```

In [3]:

```
1 # Follow-on activity:
2 # Use values stored in the eventHandle_to_eventDoss dictionary
3 # to populate a pandas dataframe, "df"
4 #
5 # MICHAEL COLLINS, 2020-09-11_2107_MDT
6
7
8 # Force seaborn to make the backgrounds of graphs white, instead of transparent
9 # Kudos to my classmate Gustavo Chavez for offering a solution to this problem
10 sns.set_style("white")
11
12 # This deletes the file specified with name strFile, if the file exists.
13 # Then it issues a plt.savefig() to save the current matplotlib to a new
14 # instance of file strFile.
15 def delete_then_plt_savefig(strFile):
16     if os.path.isfile(strFile):
17         os.remove(strFile) # Opt.: os.system("rm "+strFile)
18     plt.savefig(strFile)
19     return
20
21 # Construct a list of all unique event-level keys that will be encountered
22 print("All the event-level keys in eventHandle_to_eventDoss, as eventDoss_keys_all:")
23 keys_sorted = sorted(eventHandle_to_eventDoss.keys())
24 eventDoss_keys_all = []
25 for eh in keys_sorted:
26     eventDoss = eventHandle_to_eventDoss[eh]
27     eventDoss_keys = eventDoss.keys()
28     for k in eventDoss_keys:
29         if not k in eventDoss_keys_all:
30             eventDoss_keys_all.append(k)
31 for i, dk in enumerate(eventDoss_keys_all):
32     print("i=" + str(i) + ", " + dk)
33 print()
34
35 # Determine exactly which event-level keys should be ported into the dataframe
36 def keep_eventDoss_key(key):
37     if key in ['event_handle']:
38         return False
39     return True
40 eventDoss_keys_use = [k for k in eventDoss_keys_all if keep_eventDoss_key(k)]
41
42 # Load the pandas dataframe with desired data from eventHandle_to_eventDoss
43 df = pd.DataFrame.from_dict(eventHandle_to_eventDoss, orient='index', columns=eventDoss_keys_use).fillna(0)
44 print("df {initial values, based on eventHandle_to_eventDoss}... ")
45 display(df)
46 print()
47
48 # These are names of existing (and some possible future) columns within the dataframe
49 col_genre = 'genre_abbr'
50 col_year = 'boxOfficeYear'
51 col_genreYear = "genreYear_handle"
52 col_budget = 'productionBudget_adj'
53 col_log10_budget = 'log10_productionBudget_adj'
54 col_boy_domUS_sales = 'boxOfficeYear_domUS_adj'
55 col_log10_boy_domUS_sales = 'log10_boxOfficeYear_domUS_adj'
56 col_world_gross = "total_world_gross"
57 col_log10_world_gross = "log10_total_world_gross"
58 col_symbol_size = "symbol_size"
59 col_genre_num = "genre_num"
60 col_sbRatio = "sbRatio"
61 col_log10_sbRatio = "log10_sbRatio"
62
63 # These are new columns being added to the dataframe
64 df[col_sbRatio] = df[col_world_gross]/df[col_budget]
65 df[col_log10_sbRatio] = np.log10(df[col_sbRatio])
66 df[col_log10_boy_domUS_sales] = np.log10(df[col_boy_domUS_sales])
67 df[col_log10_budget] = np.log10(df[col_budget])
68 df[col_log10_world_gross] = np.log10(df[col_world_gross])
69 budget_max = np.max(df[col_budget])
70 df[col_symbol_size] = 100*df[col_budget]/budget_max
71 u, df[col_genre_num] = np.unique(df[col_genre], return_inverse=True)
72
73 # Show the dataframe after the new columns are added
74 print("df {after adding new columns}... ")
75 display(df)
```

```

76 print()
77
78 make_IMAGE_1 = True
79 if make_IMAGE_1:
80     fig, ax = None, None
81     print("making a strip plot... ")
82     print()
83     fig, ax = plt.subplots(figsize=(10,8))
84     sns.stripplot(data=df, y=col_genre, x=col_year, order=TN_GENRE_ABBRS,
85                     alpha=0.2, jitter=0.25, dodge=True, orient='h', marker="o", s=8)
86     plt.ylabel(None, size=26)
87     plt.xlabel("Box Office Year", size=16)
88     plt.grid()
89     plt.yticks(fontsize=14)
90     plt.xlim([1975, 2025])
91     plt.xticks(np.arange(1975, 2026, 5.0))
92     plt.xticks(fontsize=14)
93     plt.title("Movies by Genre and Box Office Year", size=20)
94     plt.tight_layout()
95     delete_then_plt_savefig("DOSFLIX_Movies_by_Genre_and_boxOfficeYear.png")
96     plt.show()
97
98
99 make_IMAGE_2 = True
100 if make_IMAGE_2:
101     fig, ax = None, None
102     print("making [sales vs. boxOfficeYear] scatter plot... ")
103     print("for movies from ALL GENRES combined... ")
104     print()
105     fig, ax = plt.subplots(figsize=(10,8))
106     sc = ax.scatter(data=df, y=col_log10_boy_domUS_sales, x=col_year, c=col_genre_num,
107                      alpha=0.8, marker="o", s=col_symbol_size)
108     ax.legend(sc.legend_elements()[0], u)
109     ax.grid()
110     ax.set_ylim(ymin=3, ymax=10)
111     plt.yticks(np.arange(3, 11, 1.0))
112     plt.yticks(fontsize=14)
113     ax.set_xlim(xmin=1975, xmax=2025)
114     plt.xticks(np.arange(1975, 2026, 5.0))
115     plt.xticks(fontsize=14)
116     s_title = "ALL GENRES represented: Log10(Sales) vs. boxOfficeYear"
117     plt.title(s_title, size=24)
118     plt.ylabel('Log10(Annual_US_Sales, ADJUSTED usd)', size=18)
119     plt.xlabel('Box Office Year', size=18)
120     fig_saveat = "DOSFLIX_Annual_US_Sales_vs_boxOfficeYear,ALL_GENRES.png"
121     delete_then_plt_savefig(fig_saveat)
122     plt.show()
123     for n, grp in df.groupby(col_genre):
124         fig, ax = None, None
125         print("making [sales vs. boxOfficeYear] scatter plot")
126         print("for movies from GENRE=" + str(n) + ", specifically... ")
127         print()
128         fig, ax = plt.subplots(figsize=(10,8))
129         c_n = gAbbr_to_gIndex[n]
130         c_list = [c_n for i in range(grp.shape[0])]
131         ax.scatter(data=grp, y=col_log10_boy_domUS_sales, x=col_year,
132                     alpha=0.8, marker="o", s=col_symbol_size)
133         ax.legend(title="Genre = " + n)
134         ax.grid()
135         ax.set_ylim(ymin=3, ymax=10)
136         plt.yticks(np.arange(3, 11, 1.0))
137         plt.yticks(fontsize=14)
138         ax.set_xlim(xmin=1975, xmax=2025)
139         plt.xticks(np.arange(1975, 2026, 5.0))
140         plt.xticks(fontsize=14)
141         s_title = "GENRE=" + str(n) + ": Log10(Sales) vs. boxOfficeYear"
142         plt.title(s_title, size=24)
143         plt.ylabel('Log10(Annual_US_Sales, ADJUSTED usd)', size=18)
144         plt.xlabel('Box Office Year', size=18)
145         fig_saveat = "DOSFLIX_Annual_US_Sales_vs_boxOfficeYear,GENRE=" + str(n) + ".png"
146         delete_then_plt_savefig(fig_saveat)
147         plt.show()
148 # the plots generated using delete_then_plt_savefig, above, were
149 # stitched together into an animated GIF "manually" via the website
150 # https://ezgif.com/
151

```

```

152
153 make_IMAGE_3 = True
154 if make_IMAGE_3:
155     fig, ax = None, None
156     print("making [sales vs. budget] scatter plot")
157     print("for movies from ALL GENRES combined... ")
158     print()
159     fig, ax = plt.subplots(figsize=(10,8))
160     for n, grp in df.groupby(col_genre):
161         #     print("n = " + repr(n))
162         #     print("grp.shape[0] = " + repr(grp.shape[0]))
163         c_n = gAbbr_to_gIndex[n]
164         c_list = [c_n for i in range(grp.shape[0])]
165         ax.scatter(data=grp, y=col_log10_world_gross, x=col_log10_budget, label=n,
166                     alpha=0.8, marker=",", s=1)
167         ax.legend(title="Genre")
168         plt.ylim([3, 10])
169         plt.yticks(np.arange(3, 11, 1.0))
170         plt.yticks(fontsize=14)
171         plt.xlim([3, 9])
172         plt.xticks(np.arange(3, 10, 1.0))
173         plt.xticks(fontsize=14)
174         plt.grid()
175         plt.title("ALL GENRES represented: World Gross vs. Budget", size=24)
176         plt.ylabel('Log10(TotalWorldwideSales, usd)', size=18)
177         plt.xlabel('Log10(Production_Budget, ADJUSTED usd)', size=18)
178         delete_then_plt_savefig("DOSFLIX_worldGross_vs_ProdBudget,ALL_GENRES.png")
179         plt.show()
180
181     fig, ax = None, None
182     for n, grp in df.groupby(col_genre):
183         print("making [sales vs. budget] scatter plots")
184         print("for movies from GENRE=" + str(n) + ", specifically... ")
185         print()
186         fig, ax = plt.subplots(figsize=(10,8))
187         c_n = gAbbr_to_gIndex[n]
188         c_list = [c_n for i in range(grp.shape[0])]
189         ax.scatter(data=grp, y=col_log10_world_gross, x=col_log10_budget,
190                     alpha=0.2, marker=",", s=10)
191         ax.legend(title="Genre = " + n)
192         plt.ylim([3, 10])
193         plt.yticks(np.arange(3, 11, 1.0))
194         plt.yticks(fontsize=14)
195         plt.xlim([3, 9])
196         plt.xticks(np.arange(3, 10, 1.0))
197         plt.xticks(fontsize=14)
198         plt.grid()
199         s_title = "GENRE=" + str(n) + ": World Gross vs. Budget"
200         plt.title(s_title, size=24)
201         plt.ylabel('Log10(TotalWorldwideSales, usd)', size=18)
202         plt.xlabel('Log10(Production_Budget, ADJUSTED usd)', size=18)
203
204         fig_saveat = "DOSFLIX_worldGross_vs_ProdBudget,GENRE=" + str(n) + ".png"
205         delete_then_plt_savefig(fig_saveat)
206         plt.show()
207     # the plots generated using delete_then_plt_savefig, above, were
208     # stitched together into an animated GIF "manually" via the website
209     # https://ezgif.com/
210
211 make_IMAGE_4 = True
212 if make_IMAGE_4:
213     fig, ax = None, None
214     print("making [sbRatio vs. budget] scatter plot")
215     print("for movies from ALL GENRES combined... ")
216     print()
217     fig, ax = plt.subplots(figsize=(10,8))
218     for n, grp in df.groupby(col_genre):
219         #     print("n = " + repr(n))
220         #     print("grp.shape[0] = " + repr(grp.shape[0]))
221         c_n = gAbbr_to_gIndex[n]
222         c_list = [c_n for i in range(grp.shape[0])]
223         ax.scatter(data=grp, y=col_log10_sbRatio, x=col_log10_budget, label=n,
224                     alpha=0.8, marker=",", s=1)
225         ax.legend(title="Genre")
226         plt.ylim([-6, 3])
227         plt.yticks(np.arange(-6, 4, 1.0))

```

```

228 plt.yticks(fontsize=14)
229 plt.xlim([3, 9])
230 plt.xticks(np.arange(3, 10, 1.0))
231 plt.xticks(fontsize=14)
232 plt.grid()
233 plt.title("ALL GENRES represented: sbRatio vs. Budget", size=24)
234 plt.ylabel('Log10(TotalWorldwideSales / ProdBudget)', size=18)
235 plt.xlabel('Log10(Production_Budget, ADJUSTED usd)', size=18)
236 delete_then_plt_savefig("DOSFLIX_sbRatio_vs_ProdBudget,ALL_GENRES.png")
237 plt.show()
238
239 fig, ax = None, None
240 for n, grp in df.groupby(col_genre):
241     print("making [sbRatio vs. budget] scatter plots")
242     print("for movies from GENRE=" + str(n) + ", specifically... ")
243     print()
244     fig, ax = plt.subplots(figsize=(10,8))
245     c_n = gAbbr_to_gIndex[n]
246     c_list = [c_n for i in range(grp.shape[0])]
247     ax.scatter(data=grp, y=col_log10_sbRatio, x=col_log10_budget,
248                 alpha=0.2, marker="o", s=10)
249     ax.legend(title="Genre = " + n)
250     plt.ylim([-6, 3])
251     plt.yticks(np.arange(-6, 4, 1.0))
252     plt.yticks(fontsize=14)
253     plt.xlim([3, 9])
254     plt.xticks(np.arange(3, 10, 1.0))
255     plt.xticks(fontsize=14)
256     plt.grid()
257     s_title = "GENRE=" + str(n) + ": sbRatio vs. Budget"
258     plt.title(s_title, size=24)
259     plt.ylabel('Log10(TotalWorldwideSales / ProdBudget)', size=18)
260     plt.xlabel('Log10(Production_Budget, ADJUSTED usd)', size=18)
261
262     fig_saveat = "DOSFLIX_sbRatio_vs_ProdBudget,GENRE=" + str(n) + ".png"
263     delete_then_plt_savefig(fig_saveat)
264     plt.show()
265 # the plots generated using delete_then_plt_savefig, above, were
266 # stitched together into an animated GIF "manually" via the website
267 # https://ezgif.com/
268
269
270
271
272 # That should do it. Thanks for reading my code. -- MLC
273 print("That's all, Folks!")
274
275 do_extra_stuff = False
276 if do_extra_stuff:
277     # This is extra stuff
278     # so please don't knock me for lack of commentary
279
280     def inspect(x):
281         print(repr(type(x)) + " " + repr(x))
282         return -1
283
284     def robust_weighted_average(x_i, w_i):
285         if np.sum(w_i) == 0.0:
286             return 0.0
287         return np.average(x_i, weights=w_i)
288
289 genre_pb_avg = df.groupby(by=col_genreYear).apply(lambda gb: robust_weighted_average(gb[col_budget], gb[col_boy_domUS_sales])/gb[col_budget].shape[0])
290 print("genre_pb_avg... ")
291 display(genre_pb_avg)
292 print()
293
294 df[col_pb_avg] = df[col_genreYear].map(genre_pb_avg)
295 print("df {AFTER col_pb_avg was added to df}... ")
296 display(df)
297 print()
298
299 genre_pb_avg_sum = df.groupby(by=col_genreYear)[col_pb_avg].sum()
300 print("genre_pb_avg_sum... ")
301 display(genre_pb_avg_sum)
302 print()
303

```

```
All the event-level keys in eventHandle_to_eventDoss, as eventDoss_keys_all:  
i=0, event_handle  
i=1, genreYear_handle  
i=2, boxOfficeYear  
i=3, genre_abbr  
i=4, movie_handle  
i=5, release_date  
i=6, productionBudget_year  
i=7, productionBudget_raw  
i=8, productionBudget_mult  
i=9, productionBudget_adj  
i=10, total_world_gross  
i=11, total_domUS_gross  
i=12, boxOfficeYear_domUS_raw  
i=13, boxOfficeYear_domUS_mult  
i=14, boxOfficeYear_domUS_adj  
i=15, boxOfficeYear_domUS_tickets
```

This is the end of Project1.