**Python Part 2 – Collections**

|  |  |
| --- | --- |
| **List Methods**  Since lists are objects, they have methods. Furthermore, since they are not contiguous arrays in memory, these methods allow for easy list manipulation. Here, "l" represents an instantiated list.  l.append(*element*)   * Adds *element* to the end of the list, l   l.extend(*iterable*)   * Adds all items in *iterable* (e.g. a list) to l   l.insert(*i*, *x*)   * Inserts an element, *x*, at position *i.* * *i* is the index before which to insert *x*   l.index(*x, start, end*)   * Returns the index of the first item whose value is equal to *x* within the slice start:end * Raises ValueError if x is not found   l.remove(*element*)   * Removes the first item in the list whose value is equal to *element*   l.sort(*key=None, reverse=False*)   * Sorts the items in place (does not return a sorted list) * Key can be a function to transform elements before sorting   l.pop(*i*)   * Removes and returns the item at position *i* * If *i* isn't supplied, the last element is removed   l.reverse()   * Reverses l in place.   **Effects of List Methods**  >>> colors = ['red', 'orange', 'yellow']  >>> more\_colors = ['blue', 'violet', 'beige']  >>>  >>> # append green to colors  >>> colors.**append**('green')  >>> colors  ['red', 'orange', 'yellow', 'green']  >>>  >>> # add the contents of more\_colors  >>> colors.**extend**(more\_colors)  >>> colors  ['red', 'orange', 'yellow', 'green', 'blue', 'violet', 'beige']  >>>  >>> # insert a color at a specific position  >>> colors.**insert**(colors.**index**('blue') + 1, 'indigo')  >>> colors  ['red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet', 'beige']  >>>  >>> # remove a color based on its value  >>> colors.**remove**('beige')  >>> colors  ['red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet']  >>>  >>> # sort the colors in reverse  >>> colors.**sort**(reverse=True)  >>> colors  ['yellow', 'violet', 'red', 'orange', 'indigo', 'green', 'blue']  >>>  >>> # remove the first element and store it  >>> alpha\_last = colors.**pop**(0)  >>> colors  ['violet', 'red', 'orange', 'indigo', 'green', 'blue']  >>> alpha\_last  'yellow'  >>>  >>> # reverse colors in place  >>> colors.reverse()  >>> colors  ['blue', 'green', 'indigo', 'orange', 'red', 'violet'] | **Example 2-1**: read in arbitrary number of strings as arguments and print them in sorted order on separate lines  $ vi sortargs.py  import sys  # extract and sort args  words = sys.argv[1:]  words.sort()  for word in words:  print(word)  $ python3 ./sortargs.py yellow zoo taco alpha  alpha  taco  yellow  zoo  Why not just sort sys.argv[1:] directly?  ??  **Example 2-2**: rotate the command arguments to the left and print the resulting list  $ vi rotate.py  import sys  if len(sys.argv) < 3:  print("Usage: " + sys.argv[0] + " <multiple\_args>")  sys.exit(1)  # remove the program name from the arguments  sys.argv.pop(0)  temp = sys.argv.pop(0)  sys.argv.append(temp)  print(sys.argv)  $ python3 ./rotate.py one two three four  ['two', 'three', 'four', 'one'] |
| **List Ranges and Generation**  Bracket notation is useful if the list is short enough to be typed out. For larger lists which can be well-defined, there are tools to help generate them.  range(*begin, end, step*)   * Returns\* an **iterator** that provides **integers** (much like [::] notation, but doesn't actually return a list) * *Begin* and *step* are optional. With only *end* supplied, the list starts at 1. * Typically used to iterate through – doesn't actually return a list * Only takes integers * \*range() is actually it's own type and not a function (but don't worry about that now)   >>> # create a list  >>> list(**range(2, 5)**)  [2, 3, 4]  >>> list(**range(3)**)  [0, 1, 2]  What would list(range(100, 200, 2)) return?  ??  **List Comprehensions**  For lists which would require a loop with logic to generate, list comprehensions can be used as shorthand.  [*expression* **for** *element* **in** *iterable* **if** *condition]*   * ***expression*** – some expression defining an element of the list (usually just the element) * **for *element* in *iterable*** – a standard for loop where *element* is the current element in *iterable* * **if *condition*** – an optional condition to decide if *element* should be included in the list   >>> # create a list  >>> [x for x in range(10, 20) if x % 2 == 0]  [10, 12, 14, 16, 18] | **Example 2-3**: Print all leap years from 1900 to 2000. Note that leap years are those divisible by four except for those divisible by 100 and not 400.  $ vi leapyear1.py  # create literal list of leap years  leap\_years = [1904, 1908, 1912, 1916, 1920, 1924, 1928, 1932,  1936, 1940, 1944, 1948, 1952, 1956, 1960, 1964, 1968,  1972, 1976, 1980, 1984, 1988, 1992, 1996]  for year in leap\_years:  print(year)  $ python3 ./leapyear1.py  1904  1908  1912  1916  1920  1924  1928  1932  1936  1940  1944  1948  1952  1956  1960  1964  1968  1972  1976  1980  1984  1988  1992  1996 |
| **Comprehensions on Multiple Inputs**  Comprehensions can take multiple lists for processing to return a single list. To do this, simply use multiple for loops and include their elements in the *expression* portion of the comprehension.  >>> colors = ['red', 'green', 'blue']  >>> cars = ['sedan', 'truck', 'coup']  >>> vehicles = **[a + " " + b for a in colors for b in cars]**  >>> vehicles  ['red sedan', 'red truck', 'red coup', 'green sedan', 'green truck', 'green coup', 'blue sedan', 'blue truck', 'blue coup'] | **Example 2-4**: Do the same as above, but with a loop to programmatically determine the years.  $ vi leapyear2.py  for year in range(1900, 2000):  if (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0):  print(year)  $ python3 ./leapyear2.py  1904  1908  1912  1916  1920  1924  1928  1932  1936  1940  1944  1948  1952  1956  1960  1964  1968  1972  1976  1980  1984  1988  1992  1996  **Example 2-5**: Do the same as above, but with list comprehension  $ vi leapyear2.py  for year in [y for y in range(1900, 2000)  if (y % 4 == 0 and y % 100 != 0) or (y % 400 == 0)]:  print(year)  $ python3 ./leapyear3.py  1904  1908  1912  1916  1920  1924  1928  1932  1936  1940  1944  1948  1952  1956  1960  1964  1968  1972  1976  1980  1984  1988  1992  1996 |
| **Functional-Style List Manipulation**  Python allows for combining functions (i.e., passing functions into other functions). This is a powerful tool for manipulation of lists where other languages might use longer, iterative approaches.  **filter()**  The filter() function allows for filtering of existing lists based on defined criteria.  filter(*function, list*)   * *function* is a function **name** which takes **one** parameter, an element from list, and returns True if the element should be kept, False otherwise. * *list* is the list to be filtered * Returns the resulting, filtered iterator which can be converted with list()   **map()**  The map() function allows for the mutation of elements in an existing list based on a mutator function.  map(*function, list*)   * *function* is a function **name** which takes **one** parameter, an element from list and returns the mapped version of the element. * *list* is the list to be mapped * Returns an iterator which can be converted with list()   **Lambdas**  Nameless functions can be created and directly passed into other functions using the lambda keyword. You can also assign lambdas directly to a variable to be used later (like def).  lambda *arguments*: *expression*  >>> def sum(a, b):  ... return a + b  ...  >>> sum(2, 2)  4  >>> # redefine sum with lambda  >>> sum2 = lambda a, b: a + b  >>> sum2(2, 2)  4  (You will learn more about functional programming and lambdas in the Programming Languages course) | **Example 2-6**: Using a **for loop**, create a list from only those arguments which correspond to an existing file. Print the list.  $ vi isfile.py  import sys  import os.path  filtered = list()  for arg in sys.argv[1:]:  if os.path.isfile(arg):  filtered.append(arg)  print(filtered)  $ python3 ./isfile.py not\_a\_file ./isfile.py ./leapyear1.py rotate.py notreal  ['./isfile.py', './leapyear1.py', 'rotate.py']  **Example 2-7**: Using **filter()**, create a list from only those arguments which correspond to an existing file. Print the list.  $ vi isfile2.py  import sys  import os.path  filtered = list(filter(os.path.isfile, sys.argv[1:]))  print(filtered)  $ python3 ./isfile2.py not\_a\_file ./isfile.py ./leapyear1.py rotate.py notreal  ['./isfile.py', './leapyear1.py', 'rotate.py']  Note that we can only use os.path.isfile because it takes a single argument (the element from the list) and returns True for those we want to keep.  **Example 2-8**: Use **map()** to create a list from arguments and convert them all to lower case strings using the string method lower().  $ vi lower\_args.py  import sys  lc = list(map(lambda a: str(a).lower(), sys.argv[1:]))  print(lc)  $ python3 ./lower\_args.py I AM YELLING!  ['i', 'am', 'yelling!'] |
| **Dictionaries**  Python dictionaries are a type of collection which allows for lookup for an element based on a corresponding **key** (i.e., a *mapping*). Keys can be strings and, more importantly, created at runtime. Dictionaries are similar to *hashes* or *associative arrays*.  Dictionaries can be created with curly braces (**{}**). Commas separate key/value pairs with colons between corresponding keys and values  >>> purchase = {"name": "apple", "qty": 5, "price": 3.25}  Notice that values do not need to be of the same type.  Dictionary values can be accessed by their keys, similar to lists.  >>> purchase["name"]  'apple'  Keys are unique. If assignment occurs for an existing key, the existing value is overwritten. If the key does not exist, a new entry is created.  >>> purchase["name"] = "banana"  >>> purchase["color"] = "yellow"  >>> purchase  {'name': **'banana'**, 'qty': 5, 'price': 3.25, **'color': 'yellow'**}  Entries can be removed from a dictionary with the **pop(*key*)** method which also returns the corresponding value.  >>> purchase.pop("color")  'yellow'  >>> purchase  {'name': 'banana', 'qty': 5, 'price': 3.25}  It can be useful to deal with just the keys or values of a hash for iteration or list-like manipulation. The **keys()** and **values()** methods return an **immutable,** **iterable object** (a dictionary view) of the corresponding elements. These objects can be used for iteration or list creation (with list()).  for key in dict.keys():  # do something with key  for val in dict.values():  # do something with value | **Example 2-9**: Dictionary manipulation using a main() function  $ vi dict.py  descriptions = {  "/": "root",  "/home": "home directories",  "/bin": "trash",  "/sbin": "system/administrative binaries",  "/etc": "configuration files",  "/dev": "device files",  "/mnt": "mounted file systems",  "/var": "system data and log files"  }  def main():  some\_dir = "/bin"  print\_summary(some\_dir)  # update an existing description  descriptions["/bin"] = "system binaries"  print\_summary(some\_dir)  # add a new description  descriptions["/boot"] = "boot-related files"  print("\nPrinting directories whose description is not 'root'")  for directory in descriptions.keys():  if descriptions[directory] != "root":  print(directory)  print("\nPrinting all values")  for description in descriptions.values():  print(description)  def print\_summary(path):  print(path + " contains " + descriptions[path] + ".")  # check if this is the main program (not an import)  if \_\_name\_\_ == '\_\_main\_\_':  main()  $ python3 dict.py  /bin contains trash.  /bin contains system binaries.  Printing directories whose description is not 'root'  /etc  /bin  /boot  /dev  /mnt  /sbin  /var  /home  Printing all values  configuration files  system binaries  boot-related files  root  device files  mounted file systems  system/administrative binaries  system data and log files  home directories  Why are the printouts out of order?  ?? |
| **Warning:** Since dictionaries use a hashing algorithm to map keys to values, the order of these mappings *cannot be guaranteed!*This means dictionaries cannot be sliced.  The collections.OrderedDict type can be used as a drop-in replacement for dictionaries as they have all the same methods but retain the order of which keys/values were inserted. However, since not all programs use OrderedDicts, it is good to be comfortable using regular dictionaries. |
| **Strings**  Systems programming primarily deals with text parsing and manipulation. For this reason, string methods are invaluable. Recall that strings are considered immutable sequences, so functions/operations that work on immutable lists can also be used on strings (e.g., len() and "in"). Here, "s" represents a string.  s.find(*e*, *start*, *end*)   * Returns leftmost position of *e* in s or -1 if not found. * *start* and *end* are optional parameters to slice the string. * Use rfind(*e*, *start*, *end*) to find rightmost position.   s.count(*substr*, *start*, *end*)   * Returns the number of occurrences of the string substr in s*.* * *start* and *end* are optional parameters to slice the string.   s.join(*sequence*)   * Returns the concatenated string of every element in sequence with s between each element. * Note that s is effectively the separator.   s.replace(*search, replace, n*)   * Returns a copy of s with every occurrence of search replaced with replace up to a maximum of n times (optional)   s.split(*e*, *n*)   * Returns a **list** of strings split from s. * *e* is the delimiter to split the string on. If not supplied, whitespace is used. * *n* is the maximum times to split. If not supplied, the string is split on every occurrence of *e.*   s.splitlines(*keep\_newlines=False*)   * Returns a **list** of strings split from s based on newline characters. * Newline characters are removed unless *keep\_newlines* is True.   s.strip(*chars*)   * Returns a copy of s with whitespace trimmed from beginning and end. * If supplied, the characters in string chars will be stripped instead of whitespace * s.rstrip(*chars*) and s.lstrip(*chars*) can be used to strip from right or left only respectively   s.lower()   * Returns a copy of s converted to lower case   s.upper()   * Returns a copy of s converted to upper case   Many more methods can be found in the Python3 documentation. | **Example 2-10**: Repeat example 2-1 (sorting arguments) but with print the out put on a single line  $ vi sortargs2.py  import sys  # extract and sort args  words = sys.argv[1:]  words.sort()  print(" ".join(words))  $ python3 ./sortargs.py yellow zoo taco alpha  alpha taco yellow zoo  **Example 2-11**: Given a single string as an argument, remove all punctuation and convert it to lower case.  $ vi sanitize.py  import sys  if len(sys.argv) != 2:  print("Usage: " + sys.argv[0] + " <string>")  sys.exit(1)  words = sys.argv[1].split()  sanitized = map(lambda w: w.strip(".,;!?'\"").lower(), words)  print(" ".join(sanitized))  $ python3 ./sanitize.py 'Hello, world!'  hello world |
| **Exercise:** Given a list of file paths as arguments, count the occurrence of each unique word and print a list of those words with their frequencies in alphabetical order. (See the output to the right.)  How do we get the words from the files? ??  How do we keep track of how many words if we don't know what words to expect beforehand (i.e., we don't know how many counters to create)? ?? | $ vi wc\_args.py  #!/usr/bin/env python3  import sys  if len(sys.argv) < 2:  print("Usage: " + sys.argv[0] + "<list\_of\_files>")  sys.exit(1)  words = dict()  for ??  try:  ??  except IOError: # catches FileNotFound and PermissionError  print("Error opening one of the files.")  sys.exit(1)  sorted\_words = ??  ??  for ??:  ??  $ python3 wc\_args.py wcin\*  a: 1  and: 1  are: 1  being: 1  data: 3  files: 2  for: 2  important: 1  in: 1  input: 1  is: 3  lot: 1  multiple: 1  of: 2  output: 1  program: 3  repeat: 1  some: 1  test: 2  tested: 1  testing: 1  the: 2  there: 1  these: 1  this: 4  use: 1  using: 1  we: 1  where: 1  will: 1  with: 1  words: 1 |
| **Note:** There are many other types of collection data types (e.g., tuples, named tuples, sets, etc) which are useful in different situations. |  |

**©2018 Rocky Slavin,** UTSA CS students may make copies for their personal use