Advanced Python Supplement

John Strickler

Version 1.0, August 2024

Table of Contents

Chapter 1: Advanced Data Handling

Objectives

- Set default dictionary values
- Count items with the Counter object
- Define named tuples
- Prettyprint data structures
- Create and extract from compressed archives
- Save Python structures to the hard drive

Deep vs shallow copying

- Normal assignments create aliases
- New objects are shallow copies
- Use the `copy.deepcopy` module for deep copies *

Consider the following code:

```
colors = ['red', 'blue', 'green']
c1 = colors
```

The assignment to variable `c1` does not create a new object; `c1` is another name that is *bound* to the same list object as the variable `colors`.

To create a new object, you can either use the list constructor list(), or use a slice which contains all elements:

```
c2 = list(colors)
c3 = colors[::]
```

In both cases, `c2` and `c3` are each distinct objects.

However, the elements of `c2` and `c3` are not copied, but are still bound to the same objects as the elements of `colors`.

For example:

```
data1 = [ [1, 2, 3], [4, 5, 6] ]
d1 = list(data)
d1[0].append(99)
print(data1)
```

This will show that the first element of data1 contains the value 99, because `data1[0]` and `d1[0]` are both bound to the same object. To do a *deep* (recursive) copy, use the `deepcopy` function in the `copy` module.

Example

deep_copy.py

```python
import copy

data = [
    [1, 2, 3],
    [4, 5, 6],
]

d1 = data # Bind d1 to same object as data
d2 = list(data) # Make shallow copy of data and store in d2
d3 = copy.deepcopy(data) # Make deep copy of data and store in d3

d1.append("d1")  # Append to d1 (same as appending to data)
d1[0].append(50) # Append to first element of d1 (same first element as data)

d2.append("d2")   # Append to d2 (does not affect data)
d2[0].append(99)  # Append to first element of d2 (same first element as data and d1)

d3.append("d3")   # Append to d3 (does not affect data)
d3[0].append(500) # Append to first element of d3 (does not affect data, d1, or d2)

print("data:", data, id(data))
print("d1:", d1, id(d1))
print("d2:", d2, id(d2))
print("d3:", d3, id(d3))
print()

print("id(d1[0]):", id(d1[0]))
print("id(d2[0]):", id(d2[0]))
print("id(d3[0]):", id(d3[0]))
```

deep_copy.py

```
data: [[1, 2, 3, 50, 99], [4, 5, 6], 'd1'] 4483340672
d1: [[1, 2, 3, 50, 99], [4, 5, 6], 'd1'] 4483340672
d2: [[1, 2, 3, 50, 99], [4, 5, 6], 'd2'] 4483340800
d3: [[1, 2, 3, 500], [4, 5, 6], 'd3'] 4483340736

id(d1[0]): 4483340608
id(d2[0]): 4483340608
id(d3[0]): 4483341056
```

Default dictionary values

- Use `defaultdict` from the `collections` module
- Specify function that provides default value
- Good for counting, datasets

Normally, when you use an invalid key with a dictionary, it raises a `KeyError`. The `defaultdict` class in the `collections` module allows you to provide a default value, so there will never be a `KeyError`. You will provide a function that returns the default value. A lambda function can be used for the function.

Example

defaultdict_ex.py

```
from collections import defaultdict

dd = defaultdict(lambda: 0)  # create default dict with function that returns 0

dd['spam'] = 10  # assign some values to the dict
dd['eggs'] = 22

print(dd['spam'])  # print values
print(dd['eggs'])
print(dd['foo'])  # missing key 'foo' invokes function and returns 0

print('-' * 60)

fruits = ["pomegranate", "cherry", "apricot", "date", "apple",
"lemon", "kiwi", "orange", "lime", "watermelon", "guava",
"papaya", "fig", "pear", "banana", "tamarind", "persimmon",
"elderberry", "peach", "blueberry", "lychee", "grape" ]

fruit_info = defaultdict(list)

for fruit in fruits:
    first_letter = fruit[0]
    fruit_info[first_letter].append(fruit)

for letter, fruits in sorted(fruit_info.items()):
    print(letter, fruits)
```

defaultdict_ex.py

```
10
22
0
------------------------------------------------------------
a ['apricot', 'apple']
b ['banana', 'blueberry']
c ['cherry']
d ['date']
e ['elderberry']
f ['fig']
g ['guava', 'grape']
k ['kiwi']
l ['lemon', 'lime', 'lychee']
o ['orange']
p ['pomegranate', 'papaya', 'pear', 'persimmon', 'peach']
t ['tamarind']
w ['watermelon']
```

Counting with Counter

- Use `collections.Counter`
- Default value is 0
- Initialize or increment elements

For ease in counting, `collections` provides a `Counter` object. This is essentially a `defaultdict` whose default value is zero. Thus, you can just increment the value for any key, whether it's been seen before or no.

In addition, you can initialize a `Counter` object with any iterable.

Example

count_with_counter.py

```
from collections import Counter

with open("../DATA/breakfast.txt") as breakfast_in:
    foods = [line.rstrip() for line in breakfast_in]  # create list of foods with EOL removed from line

counts = Counter(foods)  # initialize Counter object with list of foods

for item, count in counts.items():  # iterate over results
    print(item, count)
print()

print(f"{counts.most_common(4) = }")
```

count_with_counter.py

```
spam 34
Lucky Charms 4
eggs 3
oatmeal 1
sausage 4
upma 3
poha 4
ackee and saltfish 4
bacon 6
pancakes 2
idli 7
dosas 4
waffles 2
crumpets 1

counts.most_common(4) = [('spam', 34), ('idli', 7), ('bacon', 6), ('Lucky Charms', 4)]
```

Named Tuples

- In `collections` module
- Like `tuple`, but each element has a name
- Use either one
  - `tuple.name`
  - `tuple[n]`

A *named tuple* is a tuple where each element has a name, and can be accessed via the name in addition to the normal access by index. Thus, if `p` were a named tuple representing a point, you could say `p.x` and `p.y`, or you could say `p[0]` and `p[1]`.

A named tuple is created with the `namedtuple` class in the `collections` module. You are essentially creating a new class that inherits from `tuple`.

Pass the name of the new named tuple followed by a string containing the individual field names separated by spaces. `namedtuple` returns a new class containing the specified fields. It must be initialized with values for all fields. Being a tuple, it may not be altered once created.

Convert a named tuple into a dictionary with the `._asdict()` method.

A named tuple is the closest thing Python has to a C struct.

Example

named_tuples.py

```
from collections import namedtuple
from pprint import pprint

Knight = namedtuple('Knight', 'name title color quest comment') # create named tuple
class with specified fields (could also provide fieldnames as iterable)

k = Knight('Bob', 'Sir', 'green', 'whirled peas', 'Who am i?') # create named tuple
instance (must specify all fields)

print(k.title, k.name) # can access fields by name...
print(k[1], k[0]) # ...or index
print()

knights = [] # initialize list for knight data
with open('../DATA/knights.txt') as knights_in:
    for raw_line in knights_in:
        # strip \n then split line into fields
        name, title, color, quest, comment = raw_line.rstrip().split(':')
        # create instance of Knight namedtuple
        knight = Knight(name, title, color, quest, comment)
        # add tuple to list
        knights.append(knight)

for knight in knights: # iterate over list of knights
    print(f'{knight.title} {knight.name}: {knight.color}')
print()
pprint(knights)
```

named_tuples.py

```
Sir Bob
Sir Bob

King Arthur: blue
Sir Galahad: red
Sir Lancelot: blue
Sir Robin: yellow
Sir Bedevere: red, no blue!
Sir Gawain: blue

[Knight(name='Arthur', title='King', color='blue', quest='The Grail', comment='King of
the Britons'),
 Knight(name='Galahad', title='Sir', color='red', quest='The Grail', comment="'I could
handle some more peril'"),
 Knight(name='Lancelot', title='Sir', color='blue', quest='The Grail', comment='"It\'s
too perilous!"'),
 Knight(name='Robin', title='Sir', color='yellow', quest='Not Sure', comment='He boldly
ran away'),
 Knight(name='Bedevere', title='Sir', color='red, no blue!', quest='The Grail',
comment='AARRRRRRRGGGGHH'),
 Knight(name='Gawain', title='Sir', color='blue', quest='The Grail', comment='none')]
```

Alternatives to collections.namedtuple

There are several classes or packages for auto-generating a class or class-like object:

- `typing.NamedTuple`
- `dataclasses.dataclass`
- `pydantic`[1]
- `attrs`[1]

[1] These packages must be downloaded from **PyPI**, the Python package repository.

Table 1. Comparison of tuples and data class generators

	tuple	namedtuple	typing. NamedTuple	dataclass	pydantic	attrs
Writable	N	N	N	Y	Y	Y
Validation	N	N	N	N[2]	Y	N[2]
Iterable	Y	Y	Y	N	N	N
Indexable	Y	Y	Y	N	N	N
Methods	N	N	Y[1]	Y	Y	Y
Type hints	N	N	Y	Y	Y	Y

[1] Only when subclassing
[2] Could be added manually

Printing data structures

- Default representation of data structures is ugly
- `pprint` makes structures human friendly
- Use `pprint.pprint()`
- Useful for debugging

When debugging data structures, the `print()` function is not so helpful. A complex data structure is just printed out all jammed together, one element after another, with ony a space between elements. This can be very hard to read.

The `pprint` (pretty print) module provides the `pprint()` function, which will analyze a structure and print it out with indenting and newlines. This makes the data easier to read.

You can customize the output with named parameters. See the following table for details.

Table 2. `pprint()` parameters

Parameter	Description	Default Value
`stream`	Write data to stream (stdout if None)	`None`
`indent`	How many spaces to indent each level	`1`
`width`	Only use specified number of columns	`80`
`depth`	Only display specied levels of data	`None`
`compact`	Try to display data more compactly	`False`
`sort_dicts`	Display dictionaries sorted by keys	`True`
`underscore_numbers`	Add underscores to numbers for readability	`False`

Example

pretty_printing.py

```
from pprint import pprint

struct = {  # nested data structure
    'epsilon': [
        ['a', 'b', 'c'], ['d', 'e', 'f']
    ],
    'theta': {
        'red': 55,
        'blue': [8, 98, -3],
        'purple': ['Chicago', 'New York', 'L.A.'],
    },
    'alpha': ['g', 'h', 'i', 'j', 'k'],
    'gamma': [39029384, 3827539203, 94838402, 249398063],
}

print('Without pprint (normal output:')
print(struct)  # print normally
print()

print('With pprint:')
pprint(struct)  # pretty-print
print()

print('With pprint (sort_dicts=False):')
pprint(struct, sort_dicts=False)  # Leave dictionary in default order
print()

print('With pprint (depth=2):')
pprint(struct, depth=2)  # only print top two levels of structure
print()

print('With pprint (width=40):')
pprint(struct, width=40)  # set display width
print()

print('With pprint (underscore_numbers=True):')
pprint(struct, underscore_numbers=True)  # Put underscores in large numbers for
readability
```


pretty_printing.py

```
Without pprint (normal output:
{'epsilon': [['a', 'b', 'c'], ['d', 'e', 'f']], 'theta': {'red': 55, 'blue': [8, 98, -3],
'purple': ['Chicago', 'New York', 'L.A.']}, 'alpha': ['g', 'h', 'i', 'j', 'k'], 'gamma':
[39029384, 3827539203, 94838402, 249398063]}

With pprint:
{'alpha': ['g', 'h', 'i', 'j', 'k'],
 'epsilon': [['a', 'b', 'c'], ['d', 'e', 'f']],
 'gamma': [39029384, 3827539203, 94838402, 249398063],
 'theta': {'blue': [8, 98, -3],
           'purple': ['Chicago', 'New York', 'L.A.'],
           'red': 55}}

With pprint (sort_dicts=False):
{'epsilon': [['a', 'b', 'c'], ['d', 'e', 'f']],
 'theta': {'red': 55,
           'blue': [8, 98, -3],
           'purple': ['Chicago', 'New York', 'L.A.']},
 'alpha': ['g', 'h', 'i', 'j', 'k'],
 'gamma': [39029384, 3827539203, 94838402, 249398063]}

With pprint (depth=2):
{'alpha': ['g', 'h', 'i', 'j', 'k'],
 'epsilon': [[...], [...]],
 'gamma': [39029384, 3827539203, 94838402, 249398063],
 'theta': {'blue': [...], 'purple': [...], 'red': 55}}

With pprint (width=40):
{'alpha': ['g', 'h', 'i', 'j', 'k'],
 'epsilon': [['a', 'b', 'c'],
             ['d', 'e', 'f']],
 'gamma': [39029384,
           3827539203,
           94838402,
           249398063],
 'theta': {'blue': [8, 98, -3],
           'purple': ['Chicago',
                      'New York',
                      'L.A.'],
           'red': 55}}

With pprint (underscore_numbers=True):
{'alpha': ['g', 'h', 'i', 'j', 'k'],
 'epsilon': [['a', 'b', 'c'], ['d', 'e', 'f']],
 'gamma': [39_029_384, 3_827_539_203, 94_838_402, 249_398_063],
```



```
  'theta': {'blue': [8, 98, -3],
            'purple': ['Chicago', 'New York', 'L.A.'],
            'red': 55}}
```

Zipped archives

- import `zipfile` for zipped files
- Get a list of files
- Extract files

The `zipfile` module provides the `ZipFile` class which allows you to read and write to zipped archives. In either case you first create a `ZipFile` object, specifying the name of the zip file.

Reading zip files

Create an instance of `ZipFile`, specifying the path to the zip file.

```
To get a list of members (contained files), use ((`ZIPFILE.namelist()`)).
```

```
((`ZIPFILE.getinfo("member-name")`)) will retrieve metadata about the member as a ZipInfo
object.
```

To extract a member, use `ZIPFILE.extract("member-name")`. To read the data from a member without extracting it, use `ZIPFILE.read("member-name")`. When you read member data, is is read in binary mode, so a text file will be read in as a `bytes` object. Use `.decode()` on the bytes object to get a normal Python string.

The `tarfile` module will read and write compressed or uncompressed **tar** files.

Example

zipfile_read.py

```
from zipfile import ZipFile

# read & extract from zip file
zip_in = ZipFile("../DATA/textfiles.zip")  # Open zip file for reading
print(zip_in.namelist())  # Print list of members in zip file
tyger_text = zip_in.read('tyger.txt').decode()  # Read (raw binary) data from member and convert from bytes to string
print(tyger_text[:100], '\n')
zip_in.extract('parrot.txt')  # Extract member
```

zipfile_read.py

```
['fruit.txt', 'parrot.txt', 'tyger.txt', 'spam.txt']
           The Tyger

Tyger! Tyger! burning bright
In the forests of the night,
What immortal hand o
```

Writing to zip files

To create a new zip archive, create an instance of `ZipFile`, specifying mode 'w'. Specify `ZIP_DEFLATED` (normal compression) as the compression type.

Add files to the zip archive with the `write()` method on the `ZipFile` object.

Example

zipfile_write.py

```
from zipfile import ZipFile, ZIP_DEFLATED
import os.path

file_names = ["parrot.txt", "tyger.txt", "knights.txt", "alice.txt", "poe_sonnet.txt",
"spam.txt"]
file_folder = "../DATA"

zipfile_name = "example.zip"

# creating new, empty, zip file
zip_out = ZipFile(zipfile_name, mode="w", compression=ZIP_DEFLATED)  # Create new zip
file

# add files to zip file
for file_name in file_names:
    file_path = os.path.join(file_folder, file_name)
    zip_out.write(file_path, file_name)  # Add member to zip file
zip_out.close()

# list files in zip
zip_in = ZipFile(zipfile_name)
print("Files in archive:")
print(zip_in.namelist())
```

Making archives with **shutil**

- Create archive from folder
- Simpler than `zipfile.ZipFile`

The `shutil` module makes it easy to create an archive of an entire folder. It will create archives in the following format: **zip**, **tar**, **gztar**, **bztar**.

To create an archive, call `shutil.make_archive()`. The arguments are:

1. Base name of the output file
2. Format (one of "zip", "tar", "gztar", "bztar", or "xztar").
3. Folder to be archived (current folder if omitted)

Example

shutil_make_archive.py

```
import shutil
import os

folder = '../DATA'
archive_name = "datafiles"

for archive_type in 'zip', 'gztar':
    shutil.make_archive(archive_name, archive_type, folder)
```

Serializing Data with **pickle**

- Use the `pickle` module
- Save to file
- Transmit over network

Serializing data means taking a data structure and transforming it so it can be written to a file or other destination, and later read back into the same data structure.

Python uses the `pickle` module for data serialization.

To create pickled data, use either `pickle.dump()` or `pickle.dumps()`. Both functions take a data structure as the first argument. `pickle.dumps()` returns the pickled data as a `bytes` object. `pickle.dump()` writes the data to a file-like object which has been specified as the second argument. The file-like object must be opened for writing.

To read pickled data, use `pickle.load()`, which takes a file-like object that has been open for writing, or `pickle.loads()` which reads from a string. Both functions return the original data structure that had been pickled.

Remember to open pickle files in binary mode.

Example

pickling.py

```
import pickle
from pprint import pprint

# some data structures
airports = {
    'RDU': 'Raleigh-Durham', 'IAD': 'Dulles', 'MGW': 'Morgantown',
    'EWR': 'Newark', 'LAX': 'Los Angeles', 'ORD': 'Chicago'
}

colors = [
    'red', 'blue', 'green', 'yellow', 'black',
    'white', 'orange', 'brown', 'purple'
]

values = [
    3/7, 1/9, 14.5
]

data = [  # list of data structures
    colors,
    airports,
    values,
]

print("BEFORE:")
pprint(data)
print('-' * 60)



with open('../TEMP/pickled_data.pkl', 'wb') as pkl_out:  # open pickle file for writing in binary mode
    pickle.dump(data, pkl_out)  # serialize data structures to pickle file

with open('../TEMP/pickled_data.pkl', 'rb') as pkl_in:  # open pickle file for reading in binary mode
    pickled_data = pickle.load(pkl_in)  # de-serialize pickle file back into data structures

print("AFTER:")
pprint(pickled_data)  # view data structures
```

pickling.py

```
BEFORE:
[['red',
  'blue',
  'green',
  'yellow',
  'black',
  'white',
  'orange',
  'brown',
  'purple'],
 {'EWR': 'Newark',
  'IAD': 'Dulles',
  'LAX': 'Los Angeles',
  'MGW': 'Morgantown',
  'ORD': 'Chicago',
  'RDU': 'Raleigh-Durham'},
 [0.42857142857142855, 0.1111111111111111, 14.5]]
------------------------------------------------------------
AFTER:
[['red',
  'blue',
  'green',
  'yellow',
  'black',
  'white',
  'orange',
  'brown',
  'purple'],
 {'EWR': 'Newark',
  'IAD': 'Dulles',
  'LAX': 'Los Angeles',
  'MGW': 'Morgantown',
  'ORD': 'Chicago',
  'RDU': 'Raleigh-Durham'},
 [0.42857142857142855, 0.1111111111111111, 14.5]]
```

Chapter 1 Exercises

Exercise 1-1 (count_ext.py)

Start with the existing script `count_ext.py`.

Add code to count the number of files with each extension in a file tree.

The script will take the starting folder as a command line argument, and then display the total number of files with each distinct file extension that it finds. Files with no extension should be skipped. Use a `Counter` object to do the counting.

> Use `os.path.splitext()` to split the filename into a **path**,**extension** tuple.

Exercise 1-2 (prestuple.py, save_potus_info.py, read_potus_info.py)

Part A

Create a module named `prestuple` which defines a named tuple `President`, with fields **term**, **lastname**, **firstname**, **birthstate**, and **party**.

Part B

Write a script that uses the `csv` module to read the data from `presidents.csv` into a list of `President` named tuples. The script can import the `President` named tuple from the `prestuple` module.

Use the `pickle` module to write the list out to a file named **potus.pkl**.

Part C

Write a script to open `potus.pkl`, and restore the data back into an array.

Then loop through the array and print out each president's first name, last name, and party.

> The `prestuple` module is not needed for this script.

Exercise 1-3 (make_zip.py)

Write a script which creates a zip file containing `save_potus_info.py`, `read_potus_info.py`, and `potus.pkl`.

> Use `zipfile.ZipFile`

Chapter 2: Container Classes

Objectives

- Recap builtin container types
- Discover extra container types in standard library
- Create custom variations of container types

Container classes

- Contain all elements in memory objects
- Elements are objects or key/object pairs
- Have a length
- Are indexable and iterable

Container classes are objects that can hold multiple objects. All of the objects contained are kept in memory. Containers can be indexed, and can be iterated over. All containers have a length, which is the number of elements.

If `len(container)` is 0, the Boolean value of the container is False; otherwise the container is True.

Builtin containers

- `list`, `tuple` *array-like*
- `dict` *dictionary*
- `set`, `frozenset` *set of values*
- `str` *sequence of characters*
- `bytes` *sequence of bytes*

Several container types are builtin. All types have a length (available through the `len()` function).

Array-like

A `list` is a dynamic array, while a `tuple` is more like a struct or record. Both can be indexed or sliced.

Mapping types

The `dict` type is a dictionary of key/value pairs. In some languages, dictionaries are called hashes, or even more exotic names. Dictionaries are dynamic. Keys must be unique.

The `set` type contains unique values. Elements may be added to and deleted from a set. A `frozenset` is a readonly set.

Strings and bytes

A `str` object is an array of Unicode characters, used for any kind of text.

A `bytes` object is an array of bytes. This can be used for UTF or other encodings, and is also used for "binary" (non-text) data. If an element of a bytes object corresponds to the Unicode value of a printable character, it is displayed as the character when printed to the screen.

> Dictionary keys and set elements must be *immutable* (technically *hashable*).

Example

builtin_containers.py

source,python

```
greek_list = ['alpha', 'beta', 'gamma', 'eta', 'alpha', 'omega', 'zeta']
address = ('123 Elm Street', 'Toledo', 'Ohio')
greek_dict = {'alpha': 5, 'beta': 10, 'gamma': 15}
greek_set = {'alpha', 'beta', 'gamma', 'eta', 'zeta'}

print(f"{greek_list[0] = }")
print(f"{address[0] = }")
print(f"{greek_dict['alpha'] = }")
print(f"{greek_list[-1] = }")
print(f"{greek_list[:3] = }")
print(f"{greek_list[3:] = }")
print(f"{greek_list[2:5] = }")
print(f"{greek_list[-2:] = }")
print(f"{'alpha' in greek_list = }")
print(f"{'gamma' in greek_list = }")
print(f"{'zeta' in greek_set = }")
print(f"{len(greek_list) = }")
print(f"{len(address) = }")
print(f"{len(greek_dict) = }")
print(f"{len(greek_set) = }")
print(f"{greek_list.count('alpha') = }")
```

builtin_containers.py

```
greek_list[0] = 'alpha'
address[0] = '123 Elm Street'
greek_dict['alpha'] = 5
greek_list[-1] = 'zeta'
greek_list[:3] = ['alpha', 'beta', 'gamma']
greek_list[3:] = ['eta', 'alpha', 'omega', 'zeta']
greek_list[2:5] = ['gamma', 'eta', 'alpha']
greek_list[-2:] = ['omega', 'zeta']
'alpha' in greek_list = True
'gamma' in greek_list = True
'zeta' in greek_set = True
len(greek_list) = 7
len(address) = 3
len(greek_dict) = 3
len(greek_set) = 5
greek_list.count('alpha') = 2
```

Containers in the standard library

- Many containers in the `collections` module
- Variations of lists, tuples, and dicts

The `collections` module provides several additional container types. In general, these are variations on lists, tuples, and dicts.

Counter

A **Counter** is a dict with a default value of 0. There are two ways to use it.

- You can initialize a `Counter` object with any iterable, and it will count those values.
- Whether or not the `Counter` is initialized, you can increment the value for a given key without first checking to see whether the key exists.

In addition, counter provides a method `most_common()`, which returns the *n* most common elements counted.

Example

count_with_counter.py

```
from collections import Counter

with open("../DATA/breakfast.txt") as breakfast_in:
    foods = [line.rstrip() for line in breakfast_in]  # create list of foods with EOL
removed from line

counts = Counter(foods)  # initialize Counter object with list of foods

for item, count in counts.items():  # iterate over results
    print(item, count)
print()

print(f"{counts.most_common(4) = }")
```

count_with_counter.py

```
spam 34
Lucky Charms 4
eggs 3
oatmeal 1
sausage 4
upma 3
poha 4
ackee and saltfish 4
bacon 6
pancakes 2
idli 7
dosas 4
waffles 2
crumpets 1

counts.most_common(4) = [('spam', 34), ('idli', 7), ('bacon', 6), ('Lucky Charms', 4)]
```

defaultdict

A **defaultdict** is a dict that provides a default value for missing keys. When the defaultdict is created, you pass in a function that provides that default value. If you need a constant value, such as 0 or "", you can use a lambda expression for the function.

Example

defaultdict_ex.py

```
from collections import defaultdict

dd = defaultdict(lambda: 0)  # create default dict with function that returns 0

dd['spam'] = 10  # assign some values to the dict
dd['eggs'] = 22

print(dd['spam'])  # print values
print(dd['eggs'])
print(dd['foo'])  # missing key 'foo' invokes function and returns 0

print('-' * 60)

fruits = ["pomegranate", "cherry", "apricot", "date", "apple",
"lemon", "kiwi", "orange", "lime", "watermelon", "guava",
"papaya", "fig", "pear", "banana", "tamarind", "persimmon",
"elderberry", "peach", "blueberry", "lychee", "grape" ]

fruit_info = defaultdict(list)

for fruit in fruits:
    first_letter = fruit[0]
    fruit_info[first_letter].append(fruit)

for letter, fruits in sorted(fruit_info.items()):
    print(letter, fruits)
```

defaultdict_ex.py

```
10
22
0
------------------------------------------------------------
a ['apricot', 'apple']
b ['banana', 'blueberry']
c ['cherry']
d ['date']
e ['elderberry']
f ['fig']
g ['guava', 'grape']
k ['kiwi']
l ['lemon', 'lime', 'lychee']
o ['orange']
p ['pomegranate', 'papaya', 'pear', 'persimmon', 'peach']
t ['tamarind']
w ['watermelon']
```

namedtuple

A **namedtuple** is a tuple with specified field names. In addition to accessing fields as *tuple*[0], you can access them as *tuple*.*field_name*. Named tuples map very well to C **structs**.

Example

named_tuples.py

```
from collections import namedtuple
from pprint import pprint

Knight = namedtuple('Knight', 'name title color quest comment') # create named tuple class with specified fields (could also provide fieldnames as iterable)

k = Knight('Bob', 'Sir', 'green', 'whirled peas', 'Who am i?') # create named tuple instance (must specify all fields)

print(k.title, k.name) # can access fields by name...
print(k[1], k[0]) # ...or index
print()

knights = [] # initialize list for knight data
with open('../DATA/knights.txt') as knights_in:
    for raw_line in knights_in:
        # strip \n then split line into fields
        name, title, color, quest, comment = raw_line.rstrip().split(':')
        # create instance of Knight namedtuple
        knight = Knight(name, title, color, quest, comment)
        # add tuple to list
        knights.append(knight)

for knight in knights: # iterate over list of knights
    print(f'{knight.title} {knight.name}: {knight.color}')
print()
pprint(knights)
```

named_tuples.py

```
Sir Bob
Sir Bob

King Arthur: blue
Sir Galahad: red
Sir Lancelot: blue
Sir Robin: yellow
Sir Bedevere: red, no blue!
Sir Gawain: blue

[Knight(name='Arthur', title='King', color='blue', quest='The Grail', comment='King of
the Britons'),
 Knight(name='Galahad', title='Sir', color='red', quest='The Grail', comment="'I could
handle some more peril'"),
 Knight(name='Lancelot', title='Sir', color='blue', quest='The Grail', comment='"It\'s
too perilous!"'),
 Knight(name='Robin', title='Sir', color='yellow', quest='Not Sure', comment='He boldly
ran away'),
 Knight(name='Bedevere', title='Sir', color='red, no blue!', quest='The Grail',
comment='AARRRRRRRGGGGHH'),
 Knight(name='Gawain', title='Sir', color='blue', quest='The Grail', comment='none')]
```

deque

A **deque** (pronounced "deck") is a double-ended queue. It is optimized for inserting and removing from the ends, and is much more efficient than a **list** for this purpose. The deque can be initialized with any iterable.

Example

deque_ex.py

```
from collections import deque

d = deque()  # Create an empty deque
for c in 'abcdef':
    d.append(c)   # Append to the deque
print(f"{d = }")

for c in 'ghijkl':
    d.appendleft(c)  # Prepend to the deque
print(f"{d = }")

d.extend('mno')  # Extend the deque at the end one letter at a time
print(f"{d = }")

d.extendleft('pqr')  # Extend the deque at the beginning one letter at a time
print(f"{d = }")

print(d[9])
print(f"{d[9] = }")
print(f"{d.pop() = }")
print(f"{d.popleft() = }")

print(f"{d = }")
```

deque_ex.py

```
d = deque(['a', 'b', 'c', 'd', 'e', 'f'])
d = deque(['l', 'k', 'j', 'i', 'h', 'g', 'a', 'b', 'c', 'd', 'e', 'f'])
d = deque(['l', 'k', 'j', 'i', 'h', 'g', 'a', 'b', 'c', 'd', 'e', 'f', 'm', 'n', 'o'])
d = deque(['r', 'q', 'p', 'l', 'k', 'j', 'i', 'h', 'g', 'a', 'b', 'c', 'd', 'e', 'f',
'm', 'n', 'o'])
a
d[9] = 'a'
d.pop() = 'o'
d.popleft() = 'r'
d = deque(['q', 'p', 'l', 'k', 'j', 'i', 'h', 'g', 'a', 'b', 'c', 'd', 'e', 'f', 'm',
'n'])
```

OrderedDict

An **OrderedDict** is a dictionary which preserves order. Any iteration over the dictionary's keys or values is guaranteed to be in the same order that the elements were added.

Starting with Python 3.6, all dictionaries preserve order, making `OrderedDict` obsolete. (Of course legacy code may still use them, and some existing modules in the stdlib use or return `OrderedDict`).

Emulating builtin types

- Inherit from builtin type
- Override special methods as needed
- Use super() to invoke base class’s special methods

To emulate builtin types, you can inherit from builtin classes and override special methods as needed to get the desired behavior; other special methods will work in the normal way.

When overriding the special methods, you usually want to invoke the base class’s special methods. Use the `super()`` builtin function; the syntax is:

```
super().__specialmethod__(...)
```

Table 3. Special Methods and Variables

Method or Variables	Description
`__new__(cls,…)`	Returns new object instance; Called before `__init__()`
`__init__(self,…)`	Object initializer (constructor)
`__del__(self)`	Called when object is about to be destroyed
`__repr__(self)`	Called by `repr()` builtin
`__str__(self)`	Called by `str()` builtin

`__eq__(self, other)`
`__ne__(self, other)`
`__gt__(self, other)`
`__lt__(self, other)`
`__ge__(self, other)`
`__le__(self, other)`
	Implement comparison operators ==, !=, >, <, >=, and ⇐. self is object on the left.

`__cmp__(self, other)`	Called by comparison operators if `__eq__`, etc., are not defined
`__hash__(self)`	Called by hash`()` builtin, also used by `dict`, `set`, and `frozenset` operations
`__bool__(self)`	Called by `bool()` builtin. Implements truth value (boolean) testing. If not present, `bool()` uses `len()`
`__unicode__(self)`	Called by `unicode()` builtin

`__getattr__(self, name)`
`__setattr__(self, name, value)`
`__delattr__(self, name)`
	Override normal fetch, store, and deleter

`__getattribute__(self, name)`	Implement attribute access for new-style classes
`__get__(self, instance)`	__set__(self, instance, value)
`__del__(self, instance)`	Implement descriptors
`__slots__ = variable-list`	Allocate space for a fixed number of attributes.
`__metaclass__ = callable`	Called instead of `type()` when class is created.
`__instancecheck__(self, instance)`	Return `True` if instance is an instance of class
`__subclasscheck__(self, instance)`	Return `True` if instance is a subclass of class
`__call__(self, …)`	Called when instance is called as a function.
`__len__(self)`	Called by `len()` builtin
`__getitem__(self, key)`	Implements `self[key]`
`__setitem__(self, key, value)`	Implements `self[key] = value`
`__selitem__(self, key)`	Implements `del self[key]`
`__iter__(self)`	Called when iterator is applied to container

Method or Variables	Description
`__reversed__(self)`	Called by `reversed()` builtin
`__contains__(self, object)`	Implements `in` operator

`__add__(self, other)`
`__sub__(self, other)`
`__mul__(self, other)`
`__floordiv__(self, other)`
`__mod__(self, other)`
`__divmod__(self, other)`
`__pow__(self, other[, modulo])`
`__lshift__(self, other)`
`__rshift__(self, other)`
`__and__(self, other)`
`__xor__(self, other)`
`__or__(self, other)`
	Implement binary arithmetic operators +, -, `*, /, //, %, `**, <<, >>, &, ^, and |. **self** is object on left side of expression.

`__div__(self,other)` `__truediv__(self,other)`	Implement binary division operator `/`. `__truediv__()` is called if `__future__.division` is in effect.

`__radd__(self, other)`
`__rsub__(self, other)`
`__rmul__(self, other)`
`__rdiv__(self, other)`
`__rtruediv__(self, other)`
`__rfloordiv__(self, other)`
`__rmod__(self, other)`
`__rdivmod__(self, other)`
`__rpow__(self, other)`
`__rlshift__(self, other)`
`__rrshift__(self, other)`
`__rand__(self, other)`
`__rxor__(self, other)`
`__ror__(self, other)`
	Implement binary arithmetic operators with swapped operands. (Used if left operand does not support the corresponding operation)

`__iadd__(self, other)`
`__isub__(self, other)`
`__imul__(self, other)`
`__idiv__(self, other)`
`__itruediv__(self, other)`
`__ifloordiv__(self, other)`
`__imod__(self, other)`
`__ipow__(self, other[, modulo])`
`__ilshift__(self, other)`
`__irshift__(self, other)`
`__iand__(self, other)`
`__ixor__(self, other)`
`__ior__(self, other)`
	Implement augmented (+=, -=, etc.) arithmetic operators

`__neg__(self)`
`__pos__(self)`
`__abs__(self)`
`__invert__(self)`
	Implement unary arithmetic operators -, +, abs(), and ~

Method or Variables	Description
`__oct__(self)` `__hex__(self)`	Implement `oct()` and `hex()` builtins
`__index__(self)`	Implement `operator.index()`
`__coerce__(self, other)`	Implement "mixed-mode" numeric arithmetic.

Creating list-like containers

- Inherit from `list`
- Override special methods as needed
- Commonly overridden methods
    - __getitem__()
    - __setitem__()
    - __append__()

To create a list-like container, inherit from `list`. Commonly overridden methods include `__getitem()__` and `__setitem()__`.

Example

multiindexlist.py

```
class MultiIndexList(list):  # Define new class that inherits from list

    def __getitem__(self, item):  # Redefine __getitem__ which implements []
        if isinstance(item, tuple):  # Check to see if index is tuple
            if len(item) == 0:
                raise ValueError("Tuple must be non-empty")
            else:
                tmp_list = []
                for index in item:
                    tmp_list.append(
                        super().__getitem__(index)  # Call list.__getitem__() for each
index in tuple
                    )
                return tmp_list
        else:
            return super().__getitem__(item)  # Call the normal __getitem__()


if __name__ == '__main__':
    fruits = 'banana peach nectarine fig kiwi lemon lime'.split()
    m = MultiIndexList(fruits)  # Initialize a MultiIndexList with a list
    m.append('apple')  # Add an element (works like normal list)
    m.append('mango')
    print(m)

    print(f"m[0]: {m[0]}")  # normal indexing
    print(f"m[5, 2, 0]: {m[5, 2, 0]}")  # multi-index with tuple
    print(f"m[:4]: {m[:4]}")  # normal slice
    print(f"len(m): {len(m)}")  # len() works normally
    print(f"m[5]: {m[5]}")  # get one item (normal behavior)
    print(f"m[5,]: {m[5,]}")  # get list with just one item [m[5]]
    print(f"m[:2,-2:]: {m[:2,-2:]}")  # get list with first two, last two items
    print()
    print(f"m: {m}")
    print(m)
    m.extend(['durian', 'kumquat'])
    print(m)
    print()
    for fruit in m:
        print(fruit)
```

multiindexlist.py

```
['banana', 'peach', 'nectarine', 'fig', 'kiwi', 'lemon', 'lime', 'apple', 'mango']
m[0]: banana
m[5, 2, 0]: ['lemon', 'nectarine', 'banana']
m[:4]: ['banana', 'peach', 'nectarine', 'fig']
len(m): 9
m[5]: lemon
m[5,]: ['lemon']
m[:2,-2:]: [['banana', 'peach'], ['apple', 'mango']]

m: ['banana', 'peach', 'nectarine', 'fig', 'kiwi', 'lemon', 'lime', 'apple', 'mango']
['banana', 'peach', 'nectarine', 'fig', 'kiwi', 'lemon', 'lime', 'apple', 'mango']
['banana', 'peach', 'nectarine', 'fig', 'kiwi', 'lemon', 'lime', 'apple', 'mango',
'durian', 'kumquat']

banana
peach
nectarine
fig
kiwi
lemon
lime
apple
mango
durian
kumquat
```


Using collections.abc

- `collections.abc` contains abstract base classes
- Inherit from one or more to create hybrid types

To start from scratch, you can inherit from abstract base classes defined in the `collections.abc` module. These are base classes that define required methods for various container types. In other words, inheriting from `collections.abc.MutableSequence` provides abstract methods required for making a `list` -like class.

Example

container_abc.py

```
from collections.abc import Sized, Iterator  # Abstract base classes, used similarly to interfaces in Java or C#


class BadContainer(Sized):  # This class may not be instantiated without defining `len()`
    pass


class GoodContainer(Sized):
    def __len__(self):   # This class is fine, since `Sized` requires `len()` to be implemented
        return 42


try:
    bad = BadContainer()  # Instantiating `BadContainer` raises an error.
except TypeError as err:
    print(err)
else:
    print(bad)

print()

try:
    good = GoodContainer()  # Instantiating `GoodContainer` is fine
except TypeError as err:
    print(err)
else:
    print(good)
    print(len(good))  # Builtin function `len()` works with all objects that inherit from `Sized` (due to implementation of `len())`

print()


class MyIterator(Iterator):  # ABC `Iterator` provides abstract method `next`
    data = 'a', 'b', 'c'
    index = 0

    def __next__(self):  # Must be implemented for Iterators
        if self.index >= len(self.data):
            raise StopIteration
        else:
```



```
            return_val = self.data[self.index]
            self.index += 1
            return return_val


m = MyIterator()  # Create instance of `MyIterator`
for i in m:  # Iterate over the iterator instance
    print(i)
print()

print(hasattr(m, '__iter__'))  # Check to see if `m` is iterable
```

container_abc.py

```
Can't instantiate abstract class BadContainer with abstract method __len__

<__main__.GoodContainer object at 0x10741df50>
42

a
b
c

True
```

Creating dict-like containers

- Inherit from dict
- Implement special methods
- Commonly overridden methods
    - __getitem__
    - __haskey__
    - __setitem__

To create custom dictionaries, inherit from dict and implement special methods as needed. Commonly overridden special methods include __getitem__(), __setitem__(), and __haskey__().

Example

stringkeydict.py

```
class StringKeyDict(dict):  # Create class that inherits from dict
    def __setitem__(self, key, value):  # Overwrite how values are stored in the dict via
_DICT_[_KEY_] = _VALUE_
        if isinstance(key, str):   # Make sure key is a string
            super().__setitem__(key, value)  # Use dict's setitem to set value if it is
not a key
        else:
            # Raise error if non-string key is used
            raise TypeError(f"Keys must be strings not {type(key).__name__}s")


if __name__ == '__main__':
    d = StringKeyDict(a=10, b=20)   # Create and initialize StringKeyDict instance
    for k, v in [('c', 30), ('d', 40), (1, 50), (('a', 1), 60), (5.6, 201)]:
        try:
            print(f"Setting {k} to {v}", end=' ')
            d[k] = v   # Try to add various key/value pairs
        except TypeError as err:
            print(err)  # Error raised on non-string key
        else:
            print('SUCCESS')

    print()
    print(d)
```

stringkeydict.py

```
Setting c to 30 SUCCESS
Setting d to 40 SUCCESS
Setting 1 to 50 Keys must be strings not ints
Setting ('a', 1) to 60 Keys must be strings not tuples
Setting 5.6 to 201 Keys must be strings not floats

{'a': 10, 'b': 20, 'c': 30, 'd': 40}
```


Free-form containers

- Easy to create hybrid containers
- Objects may implement any special methods
- Create list+dict, ordered set
- Be creative

There’s really no limit to the types of objects you can create. You can make hybrids that act like lists and dictionaries at the same time. Just implement the special methods required for this behavior.

A well-known example of this is the **Element** class in the lxml.etree module. An Element is a list of its children, and at the same time is a dictionary of its XML attributes:

```
e = Element(...)
child_element = e[0]
attr_value = e.get("attribute")
```

Chapter 2 Exercises

Exercise 2-1 (maxlist.py, ringlist.py)

Part A

Create a new type, `MaxList`, that will only grow to a certain size. It should raise an `IndexError` if the user attempts to add an item beyond the limit. `MaxList` should inherit from `list` (or you can start from scratch with `collections.abc.MutableSequence`).

Run the provided script `use_maxlist.py` to try out your `MaxList` class.

HINT: You will need to override the `append()`, `insert()`, and `extend()` methods; to be thorough, you would need to override `__init__()` as well, so that the initializer doesn't have more than the maximum number of items.

Part B *(for the ambitious)*

Copy `maxlist.py` to `ringlist.py` and create a `RingList` class that, once it reaches maximum size, appending to the list also removes the first element, so the list stays constant size.

Run the provided script `use_ringlist.py` to try out your `RingList` class.

Exercise 2-2 (normalstringdict.py)

Create a module which defines the class `NormalStringDict`, that only allows strings as keys *and* values. Values (but not keys) should be normalized by removing surrounding white space and converting to lower case.

Run the provided script `use_nsd.py` to try out your `NormalStringDict` class.

Index