ADVANCED

HANDS-ON PYTHON SERIES

HANDS-ON PYTHON

with Exercises, Projects, Assignments & Final Exam

MUSA ARDA

HANDS-ON PYTHON

with Exercises, Projects, Assignments & Final Exam

ADVANCED

Musa Arda

Hands-On Python with Exercises, Projects, Assignments & Final Exam: Advanced

By Musa Arda

Contents

Preface

About This Book

This book is an **in-depth** and **activity-based** introduction to the **advanced** level topics of Python programming. It follows a step-by-step practical approach by combining the theory of the language with the hands-on coding exercises including quizzes, projects, assignments and exams.

We begin by introducing the Collections module in Python. Then we cover Iterators, Generators, Date and Time Operations, Decorators and Context Managers in Python.

By the end of the book, you will learn almost all of the advanced level concepts of Python in great detail by writing thousands of lines of code. All the supplementary resources (**code files**, **quizzes**, **assignments**, **final exam** etc.) are available for download at the **GitHub** repository. The link for the repository is provided in the book.

This is the third book in Hands-On Python Series. And here is what you will find in this book:

Theory: In each topic, we will cover all the Theoretical Details with example coding.

Coding Exercises: At the end of each chapter, we will have Coding Exercise, Quizzes.

Projects: We will build projects in this book. You will learn how to apply Python concepts on real world problems.

Assignments: After each project, you will have an Assignment. These assignments will let you build the project from scratch on your own.

Final Exam: At the end of this book, you will have the Final Exam. It is a multiple-choice exam with 20 questions and a limited duration. The exam will let you to test your Python level.

About Hands-On Python Series

This is the third book in our Hands-On Python Series. And it covers the advanced level topics. So, we assume that you already know the introductory and intermediate concepts of Python programming like; Variables, Functions, Conditional Statements, Loops, Strings, Lists, Dictionaries, Tuples, Sets and Comprehensions, Exception Handling, File Operations and OOP. If you don’t feel comfortable with these topics, you are strongly recommended to finish the first two books in our series, which are the Beginner and Intermediate levels. Here, you can find them.[1]

About the Author

Musa Arda has Bachelor’s degree from Industrial Engineering in 2007, and he has been working as a Software Developer for more than 14 years. He uses many programming languages and platforms like; Python, C#, Java, JavaScript, SAP ABAP, SQL, React, Flutter and more.

He creates online learning content and writes books to share his experience and knowledge with his students. His main purpose is to combine theory and hands-on practice in his teaching resources. That’s why there are hundreds of programming exercises, quizzes, tests, projects, exams and assignments in all of his courses and books. He is dedicated to help his students to learn programming concepts in depth via a practical and exiting way.

How to Contact Us

Please feel free to get in contact with the author. To comment or ask technical questions about this book, you can send email to ***python.hands.on.book@gmail.com***.

1. Introduction

Who Is This Book For?

The goal of this book is to help students to learn Python programming language in a hands-on and project-based manner.

With its unique style of combining theory and practice, this book is for:

- people who want to learn and practice advanced concepts in Python programming
- people who are already working with Python language

What Can You Expect to Learn?

The purpose of this book is to provide you a good introduction to the advanced topics of Python programming. In general, you will gain solid programming skills and grasp the main idea of software development. In particular, here are some highlights on what you can you expect to learn with this book.

You will:

- learn & master advanced Python topics in a hands-on approach
- practice your Python knowledge with Quizzes and Coding Exercises
- build Real-World Project with Python and do Assignments related to these projects
- take the Final Exam on Python with 20 questions to assess your learning
- build Python applications with PyCharm and master it
- gain solid and profound Python Programming skills needed for a Python career

Outline of This Book

In Chapter 1, you will get the basics of the book. You will learn about this books approach to Python programming and how to get most out of it.

In Chapter 2, you will learn the Collections module in Python. Collections are specialized container datatypes providing alternatives to Python’s general-purpose containers, **dict**, **list**, **set**, and **tuple**. You will learn; ChainMap, Counter, Deque, DefaultDict, NamedTuple, OrderedDict, UserDict, UserList, and UserString.

In Chapter 3, you will learn Iterables and Iterators in Python. You will see the details of Iterator Protocol, how to loop through an Iterator, how to define custom Iterators and Infinite Iterators.

Chapter 4 is on Generators in Python. Generators allow you to define Iterators more easily and efficiently. You will define custom Generators and learn the benefits of using them.

In Chapter 5, you will learn all the details of Date and Time operations in Python, which are crucial for robust application development. You will learn the difference between Aware and Naive Objects and get the details of the classes in datetime module.

In Chapter 6, you will meet Decorators. A very important concept in Python programming. You will learn how to define a decorator, how to chain them and how to use class syntax for creating new ones.

In Chapter 7, you will learn the details of Context Managers, which are very handy tools when you need to deal with resource management in your code. You will see how to define and use a Context Manager in both class form and function form.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new URLs, email addresses, filenames, and file extensions.

Bold

Indicates new terms and important concepts to pay attention.

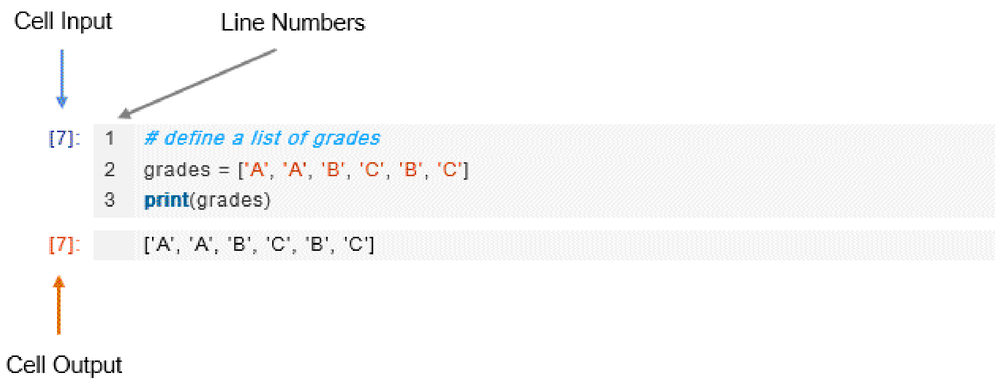
`Constant width`

Used for string literals and programming concepts within paragraphs.

`Constant width bold`

Used for program elements within paragraphs, such as variable or function names, data types, statements, and keywords.

We will write our code in the code cells. Here is an example code cell with the cell number as 7:



```
# define a list of grades
grades = ['A', 'A', 'B', 'C', 'B', 'C']
print(grades)
```

```
['A', 'A', 'B', 'C', 'B', 'C']
```

Figure 1-1: A code cell example used in this book

Using Code Examples

You can find all the supplementary resources for the book (code files, quizzes, assignments, final exam etc.) available for download at https://github.com/musaarda/python-hands-on-book-advanced.

You & This Book

This book is designed in a way that, you can learn and practice Python. At each chapter you will learn the basic concepts and how to use them with examples. Then you will have a quiz the end of the chapter. First you will try to solve the quiz questions on your own, then I will provide the solutions in detail. You will have projects after each block of core concepts. And after every project you will an assignment to test your understanding. This will be your path to learn real Python.

Before we deep dive into Python, I want to give you some tips for how you can get most out of this book:

- Read the topics carefully before you try to solve the quizzes
- Try to code yourself while you are reading the concepts in the chapters
- Try to solve quizzes by yourself, before checking the solutions
- Read the quiz solutions and try to replicate them
- Code the projects line by line with the book
- Do the assignments (seriously)
- Do not start a new chapter before finishing and solving quiz of the previous one
- Repeat the topics you fail in the Final Exam
- Learning takes time, so give yourself enough time digest the concepts

2. Collections

What are Collections

Collections are specialized container datatypes providing alternatives to Python’s general purpose built-in containers, **`dict`**, **`list`**, **`set`**, and **`tuple`**. A Container is a special-purpose object which is used to store different objects. It provides a way to access the contained objects and iterate over them.

Python provides the collections module which implements container datatypes. In this chapter we will learn different classes in the collections module. You can find the PyCharm project for this chapter in the Github Repository of this book.

Chapter Outline:

- ChainMap
- Counter
- Deque
- DefaultDict
- NamedTuple
- OrderedDict
- UserDict
- UserList
- UserString

ChainMap

A **`ChainMap`** class is provided for quickly linking a number of mappings so they can be treated as a single unit. It is often much faster than creating a new dictionary and running multiple **`update()`** calls.

Syntax:

```
class collections.ChainMap(*maps)
```

A ChainMap groups multiple dicts or other mappings together to create a single, updateable view (list of dictionaries). If no maps are specified, a single empty dictionary is provided so that a new chain always has at least one mapping.

The underlying mappings are stored in a list. That list is public and can be accessed or updated using the **maps** attribute. There is no other state.

A ChainMap incorporates the underlying mappings by reference. So, if one of the underlying mappings gets updated, those changes will be reflected in ChainMap.

All of the usual dictionary methods are supported. In addition, there is a **maps** attribute, a method for creating new subcontexts, and a property for accessing all but the first mapping:

maps:

A user updateable list of mappings. The list is ordered from first-searched to last-searched. It is the only stored state and can be modified to change which mappings are searched. The list should always contain at least one mapping.

[1]:
```
# import ChainMap class from collections module
from collections import ChainMap

#– Defining a ChainMap –#
numbers = {‘one’: 1, ‘two’: 2}
letters = {‘a’: ‘A’, ‘b’: ‘B’}

# Define the ChainMap
chain_map = ChainMap(numbers, letters)

print(chain_map)
```

[1]:
```
ChainMap({‘one’: 1, ‘two’: 2}, {‘a’: ‘A’, ‘b’: ‘B’})
```

In cell 1, we define a ChainMap object (**chain_map**) with two dictionaries. Then we print the ChainMap. As you see in the output, the result is a view of these dicts.

Accessing Keys and Values from ChainMap:

We can access the keys and values of a ChainMap by using the **keys()** and **values()** methods.

[2]:
```
#– Accessing Keys and Values from ChainMap –#
print(chain_map.keys())
print(chain_map.values())
```

[2]:
```
KeysView(ChainMap({'one': 1, 'two': 2}, {'a': 'A', 'b': 'B'}))
ValuesView(ChainMap({'one': 1, 'two': 2}, {'a': 'A', 'b': 'B'}))
```

As you see in the output of cell 2, the result of **chain_map.keys()** is a **KeysView** and the result of **chain_map.values()** is a **ValuesView**.

Accessing Individual Values with Key Names:

We can access individual values from a ChainMap by using the key name. This is exactly the same way what we do with regular dictionaries.

[3]:
```
#– Accessing Individual Values with Key Names –#
print(chain_map['one'])
print(chain_map['b'])
```

[3]:
```
1
B
```

In cell 3, we access the values of the individual items in the underlying dictionaries of the ChainMap by using the key names as: **chain_map['one']**.

Adding a New Dictionary to ChainMap:

ChainMap can contain any number of dictionaries in it. We use the built-in **new_child()** method to add new dictionaries to the ChainMap. The

new_child() method returns a new ChainMap containing a new map followed by all of the maps in the current instance. One point to note here is, the newly added dict will be placed at the beginning of the ChainMap.

[4]:
```
#– Adding a New Dictionary to ChainMap –#
variables = {‘x’: 0, ‘y’: 1}
new_chain_map = chain_map.new_child(variables)
print(‘Old:’, chain_map)
print(‘New:’, new_chain_map)
```

[4]:
```
Old: ChainMap({‘one’: 1, ‘two’: 2}, {‘a’: ‘A’, ‘b’: ‘B’})
New: ChainMap({‘x’: 0, ‘y’: 1}, {‘one’: 1, ‘two’: 2}, {‘a’: ‘A’,
‘b’: ‘B’})
```

Get the List of Mappings in ChainMap:

We use the **maps** attribute the get the list of all mappings in the ChainMap.

[5]:
```
#– Get the List of Mappings in ChainMap –#
print(chain_map.maps)
```

[5]:
```
[{‘one’: 1, ‘two’: 2}, {‘a’: ‘A’, ‘b’: ‘B’}]
```

In cell 5, we get all the mappings (dictionaries) in the **chain_map**. As you see in the output, the **maps** attribute returns a **list** type object.

Counter

A **Counter** is a dict subclass for counting hashable objects. It is a collection where elements are stored as dictionary keys and their counts are stored as dictionary values. Counts are allowed to be any integer value including zero or negative counts. The Counter class is similar to bags or multisets in other languages.

Elements are counted from an iterable or initialized from another mapping (or counter). Here are some ways we create Counter objects in Python:

[6]:
```python
from collections import Counter

# a new, empty counter
c1 = Counter()
print(c1)

# a new counter from an iterable
c2 = Counter('aabbbcddeeee')
print(c2)

# a new counter from a mapping
c3 = Counter({'orange': 6, 'red': 3, 'green': 5})
print(c3)

# a new counter from keyword args
c4 = Counter(cats=4, dogs=8)
print(c4)
```

[6]:
```
Counter()
Counter({'e': 4, 'b': 3, 'a': 2, 'd': 2, 'c': 1})
Counter({'orange': 6, 'green': 5, 'red': 3})
Counter({'dogs': 8, 'cats': 4})
```

Counter objects have a dictionary interface except that they return a zero count for missing items instead of raising a **KeyError**:

[7]:
```python
# count of existing element
```

```
c5 = Counter([‘eggs’, ‘ham’, ‘jar’, ‘ham’])
print(c5[‘ham’])

# count of a missing element is zero
print(c5[‘bacon’])
```

[7]:
```
2
0
```

Delete Elements from a Counter:

To delete elements from a Counter, we use the **del** keyword. Please keep in mind that, setting a count to zero does not remove an element from a counter.

[8]:
```
1   # – Delete Elements from a Counter – #
2   # counter entry with a zero count
3   c5[‘sausage’] = 0
4   print(c5)
5
6   # del actually removes the entry
7   del c5[‘sausage’]
8   print(c5)
```

[8]:
```
Counter({‘ham’: 2, ‘eggs’: 1, ‘jar’: 1, ‘sausage’: 0})
Counter({‘ham’: 2, ‘eggs’: 1, ‘jar’: 1})
```

As you see in cell 8, we set zero to an item which even doesn’t exist in the Counter. And Python adds that item to the Counter with zero value. In line 7, we remove the item entirely with the **del** keyword.

Counter Methods:

Counter objects support additional methods beyond those available for all dictionaries. Here are the most common methods:

elements():

Return an iterator over elements repeating each as many times as its count. Elements are returned in the order first encountered. If an element's count is less than one, **elements()** will ignore it.

[9]:
```
# – Counter Methods – #
# elements()
counter = Counter(a=1, b=2, c=0, d=-2, e=4)
sorted_elements = sorted(counter.elements())
print(sorted_elements)
```

[9]:
```
['a', 'b', 'b', 'e', 'e', 'e', 'e']
```

most_common([n]):

Return a list of the **n** most common elements and their counts from the most common to the least. If n is omitted or None, **most_common()** returns all elements in the counter. Elements with equal counts are ordered in the order first encountered:

[10]:
```
# most_common()
most_common_3 = Counter('abracadabra').most_common(3)
print(most_common_3)
```

[10]:
```
[('a', 5), ('b', 2), ('r', 2)]
```

subtract([iterable-or-mapping]):

Elements are subtracted from an iterable or from another mapping (or counter). Like **dict.update()** but subtracts counts instead of replacing them. Both inputs and outputs may be zero or negative.

[11]:
```python
# subtract()
c_1 = Counter(a=4, b=2, c=0, d=-2)
c_2 = Counter(a=1, b=2, c=3, d=4)
c_1.subtract(c_2)
print(c_1)
```

[11]:
```
Counter({‘a’: 3, ‘b’: 0, ‘c’: -3, ‘d’: -6})
```

The usual dictionary methods are available for Counter objects except for two which work differently for counters:

fromkeys(iterable):

This class method is not implemented for Counter objects.

update([iterable-or-mapping]):

Elements are counted from an *iterable* or added-in from another *mapping* (or counter). Like **dict.update()** but adds counts instead of replacing them. Also, the *iterable* is expected to be a sequence of elements, not a sequence of (key, value) pairs.

[12]:
```python
# update()
d = Counter(a=3, b=1)
d.update({‘a’: 5, ‘c’: 4})
print(d)
```

[12]:
```
Counter({‘a’: 8, ‘c’: 4, ‘b’: 1})
```

Deque

Deques are a generalization of stacks and queues (the name is pronounced “deck” and is short for “double-ended queue”). Deques support thread-safe, memory efficient appends and pops from either side of the deque with approximately the same **O(1)** performance in either direction.

Though list objects support similar operations, they are optimized for fast fixed-length operations and incur **O(n)** memory movement costs for **pop(0)** and **insert(0, v)** operations which change both the size and position of the underlying data representation.

collections.deque([iterable[, maxlen]]): Returns a new deque object initialized left-to-right (using **append()**) with data from iterable. If iterable is not specified, the new deque is empty.

maxlen: Maximum size of a deque or **None** if unbounded.

If **maxlen** is not specified or is **None**, deques may grow to an arbitrary length. Otherwise, the deque is bounded to the specified maximum length. Once a bounded length deque is full, when new items are added, a corresponding number of items are discarded from the opposite end.

[13]:
```
### Deque

from collections import deque

# Declaring the deque
q = deque([‘user’, ‘password’, ‘token’])
print(q)
```

[13]:
```
deque([‘user’, ‘password’, ‘token’])
```

In cell 13, we define a deque object by passing a list as argument. Now let’s create another one but this time we will use a string:

[14]:
```
# make a new deque with three items
d = deque(‘dqi’)
# iterate over the deque’s elements
for elem in d:
    print(elem.upper())
```

[14]:
```
D

Q

I
```

Now let’s see the contents in the deque object:

[15]:
```
# list the contents of the deque
deque_contents = list(d)
print(deque_contents)

# peek at leftmost item
print(d[0])

# peek at rightmost item
print(d[-1])
```

[15]:
```
[‘d’, ‘q’, ‘i’]

d

i
```

Here are some methods that deque objects support:

append(x):

Add x to the right side of the deque.

appendleft(x):

Add x to the left side of the deque.

[16]:
```
# add a new entry to the right side
d.append(‘j’)

```

```
# add a new entry to the left side
d.appendleft('f')

# show the representation of the deque
print(d)
```

[16]:
```
deque(['f', 'd', 'q', 'i', 'j'])
```

pop():

Remove and return an element from the right side of the deque. If no elements are present, raises an **IndexError**.

popleft():

Remove and return an element from the left side of the deque. If no elements are present, raises an **IndexError**.

[17]:
```
# return and remove the rightmost item
rightmost = d.pop()
print(rightmost)

# return and remove the leftmost item
leftmost = d.popleft()
print(leftmost)
```

[17]:
```
j
f
```

clear():

Remove all elements from the deque leaving it with length 0.

copy():

Create a shallow copy of the deque.

count(x):

Count the number of deque elements equal to x.

extend(iterable):

Extend the right side of the deque by appending elements from the iterable argument.

[18]:
```
# add multiple elements at once
d.extend('jkl')
print(d)
```

[18]:
```
deque(['d', 'q', 'i', 'j', 'k', 'l'])
```

extendleft(iterable):

Extend the left side of the deque by appending elements from iterable. Note, the series of left appends results in reversing the order of elements in the iterable argument.

[19]:
```
# extendleft() reverses the input order
d.extendleft('xyz')
print(d)
```

[19]:
```
deque(['z', 'y', 'x', 'd', 'q', 'i', 'j', 'k', 'l'])
```

index(x[, start[, stop]]):

Return the position of x in the deque (at or after index start and before index stop). Returns the first match or raises `ValueError` if not found.

insert(i, x):

Insert x into the deque at position i. If the insertion would cause a bounded deque to grow beyond `maxlen`, an `IndexError` is raised.

remove(value):

Remove the first occurrence of value. If not found, raises a **ValueError**.

rotate(n=1):

Rotate the deque n steps to the right. If n is negative, rotate to the left.

[20]:
```
# deque at the beginning
print(d)

# right rotation
d.rotate(1)
print(d)

# left rotation
d.rotate(-1)
print(d)
```

[20]:
```
deque([‘z’, ‘y’, ‘x’, ‘d’, ‘q’, ‘i’, ‘j’, ‘k’, ‘l’])
deque([‘l’, ‘z’, ‘y’, ‘x’, ‘d’, ‘q’, ‘i’, ‘j’, ‘k’])
deque([‘z’, ‘y’, ‘x’, ‘d’, ‘q’, ‘i’, ‘j’, ‘k’, ‘l’])
```

reverse():

Reverse the elements of the deque **in-place** and then return **None**.

[21]:
```
# deque at the beginning
print(‘old deque:’, d)

# reverse the elements in the deque
new_deq = d.reverse()
```

```
print('new deque:', new_deq)

# original deque after reversed()
print('old deque:', d)
```

[21]:

```
old deque: deque(['z', 'y', 'x', 'd', 'q', 'i', 'j', 'k', 'l'])
new deque: None
old deque: deque(['l', 'k', 'j', 'i', 'q', 'd', 'x', 'y', 'z'])
```

As you see in the output of cell 21, the **reverse()** method reverses the elements of the deque in-place, which means our original deque object is modified. And it returns **None**.

DefaultDict

One of the common problems with the Dictionary class in Python is the missing keys. When you try to access a key that does not exist in the dictionary you will get a KeyError. So have to handle this case whenever you need to access an element in the dictionary. Fortunately, we have **DefaultDict** class in Python. It is used to provide some default values for the key that does not exist and does not raise a **KeyError**.

DefaultDict is a subclass of the built-in dict class. It overrides one method and adds one writable instance variable. The remaining functionality is the same as for the dict class and is not documented here.

collections.defaultdict(default_factory=None, /[, ...]): Return a new dictionary-like object, DefaultDict, which is a subclass of the built-in dict class.

The first argument provides the initial value for the **default_factory** attribute; it defaults to **None**. All remaining arguments are treated the same as if they were passed to the dict constructor, including keyword arguments.

DefaultDict objects support the following method in addition to the standard dict operations:

__missing__(key):

If the **default_factory** attribute is **None**, this raises a **KeyError** exception with the key as argument.

If **default_factory** is not **None**, it is called without arguments to provide a default value for the given key, this value is inserted in the dictionary for the key, and returned.

DefaultDict objects support the following instance variable:

default_factory:

This attribute is used by the **__missing__()** method; it is initialized from the first argument to the constructor, if present, or to **None**, if absent.

[22]:
```
from collections import defaultdict

s = [('yellow', 1), ('blue', 2), ('yellow', 3), ('blue', 4), ('red', 1)]
d = defaultdict(list)
for k, v in s:
    d[k].append(v)

sorted_items = sorted(d.items())
print(sorted_items)
```

[22]:
```
[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]
```

In cell 20, we use the list type as the **default_factory**, to make it easy to group a sequence of key-value pairs into a dictionary of lists. When each key is encountered for the first time, it is not already in the mapping; so an entry is automatically created using the **default_factory** function which returns an empty list. The **list.append()** operation then attaches the value to the new list. When keys are encountered again, the look-up proceeds normally (returning the list for that key) and the **list.append()** operation adds another value to the list. This technique is simpler and faster than an equivalent technique using **dict.setdefault()**.

[23]:
```
river = 'mississippi'
dd = defaultdict(int)
```

```
for r in river:
    dd[r] += 1

s_items = sorted(dd.items())
print(s_items)
```

[23]:

```
[('i', 4), ('m', 1), ('p', 2), ('s', 4)]
```

In cell 23, we set the **default_factory** to **int**. This makes the defaultdict useful for counting (like a bag or multiset in other languages). When a letter is first encountered, it is missing from the mapping, so the **default_factory** function calls **int()** to supply a default count of zero. The increment operation then builds up the count for each letter.

NamedTuple

NamedTuples assign meaning to each position in a tuple and allow for more readable, self-documenting code. They can be used wherever regular tuples are used, and they add the ability to access fields by name instead of position index.

collections.namedtuple(typename, field_names):

Returns a new tuple subclass named **typename**. The new subclass is used to create tuple-like objects that have fields accessible by attribute lookup as well as being indexable and iterable. Instances of the subclass also have a helpful docstring (with **typename** and **field_names**) and a helpful **__repr__()** method which lists the tuple contents in a name=value format.

The **field_names** are a sequence of strings such as ['x', 'y']. Alternatively, **field_names** can be a single string with each fieldname separated by whitespace and/or commas, for example 'x y' or 'x, y'.

To understand how the NamedTuple works, let's assume we have an Employee object. Employee has id, name and age attributes.

[24]:

```
from collections import namedtuple

```

```
# Declare the namedtuple
Employee = namedtuple(‘Employee’, [‘id’, ‘name’, ‘age’])

# Add some values to the tuple
E_1 = Employee(‘111’, ‘Peter Parker’, ‘18’)
E_2 = Employee(‘222’, ‘Clark Kent’, ‘26’)

# Access using index
print(“Employee name by index is : “, end=””)
print(E_1[1])

# Access using keys
print(“Employee name using key is : “, end=””)
print(E_2.name)
```

[24]:
```
Employee name by index is : Peter Parker
Employee name using key is : Clark Kent
```

In addition to the methods inherited from tuples, named tuples support three additional methods and two attributes. To prevent conflicts with field names, the method and attribute names start with an underscore.

_make(iterable):

Class method that makes a new instance from an existing sequence or iterable.

[25]:
```
# _make()
# initialize an iterable
bat_data = [‘333’, ‘Batman’, ‘28’]
batman = Employee._make(bat_data)
print(batman)
```

[25]:
```
Employee(id=‘333’, name=‘Batman’, age=‘28’)
```

_asdict():

Return a new dict which maps field names to their corresponding values:

[26]:
```
# _asdict()
bat_dict = batman._asdict()
print(bat_dict)
```

[26]:
```
{‘id’: ‘333’, ‘name’: ‘Batman’, ‘age’: ‘28’}
```

_replace(kwargs)**:

Return a new instance of the named tuple replacing specified fields with new values:

[27]:
```
# _replace()
batman = batman._replace(id=‘777’, age=‘34’)
print(batman)
```

[27]:
```
Employee(id=‘777’, name=‘Batman’, age=‘34’)
```

_fields:

Tuple of strings listing the field names. Useful for introspection and for creating new named tuple types from existing named tuples.

[28]:
```
# _fields
print(batman._fields)
```

[28]:
```
(‘id’, ‘name’, ‘age’)
```

We can use the **_fields** attribute to create new namedtuples from existing ones:

[29]:
```
# namedtuple fields from others
Point = namedtuple('Point', ['x', 'y'])
Color = namedtuple('Color', 'red green blue')
Pixel = namedtuple('Pixel', Point._fields + Color._fields)
p = Pixel(5, 8, 128, 255, 0)
print(p)
```

[29]:
```
Pixel(x=5, y=8, red=128, green=255, blue=0)
```

OrderedDict

Ordered Dictionaries are just like regular dictionaries but have some extra capabilities relating to ordering operations. **OrderedDicts** remember the order in which the keys were inserted. They have become less important now that the built-in dict class gained the ability to remember insertion order (this new behavior became guaranteed in Python 3.7).

collections.OrderedDict([items]):

Return an instance of a dict subclass that has methods specialized for rearranging dictionary order.

popitem(last=True):

The **popitem()** method for ordered dictionaries returns and removes a (key, value) pair. The pairs are returned in LIFO order if last is true or FIFO order if false.

move_to_end(key, last=True):

Move an existing key to either end of an ordered dictionary. The item is moved to the right end if last is true (the default) or to the beginning if last is false. Raises KeyError if the key does not exist:

[30]:
```
from collections import OrderedDict

od = OrderedDict.fromkeys('abcde')
od.move_to_end('b')
print(''.join(od))
# 'acdeb'
od.move_to_end('b', last=False)
print(''.join(od))
# 'bacde'
```

[30]:
```
acdeb
bacde
```

Let's say we delete and re-insert the same key to an OrderedDict. It will push this key to the end to maintain the order of insertion of the keys.

[31]:
```
# delete and re-insert same key
d = OrderedDict()
d['x'] = 'X'
d['y'] = 'Y'
d['z'] = 'Z'

print('OrderedDict before deleting')
for key, value in d.items():
    print(key, value)

# delete the element
```

```
d.pop('x')

# re-insert the same key
d['x'] = 'X'

print('\nOrderedDict after insertion')
for key, value in d.items():
    print(key, value)
```

[31]:

```
OrderedDict before deleting
x X
y Y
z Z


OrderedDict after insertion
y Y
z Z
x X
```

UserDict

The class, **UserDict** acts as a wrapper around dictionary objects. The need for this class has been partially supplanted by the ability to subclass directly from dict; however, this class can be easier to work with because the underlying dictionary is accessible as an attribute. You can use UserDict when you want to create your own dictionary with some modified or new functionality.

collections.UserDict([initialdata]):

Class that simulates a dictionary. The instance's contents are kept in a regular dictionary, which is accessible via the data attribute of UserDict instances. If **initialdata** is provided, data is initialized with its contents; note that a reference to **initialdata** will not be kept, allowing it to be used for other purposes.

In addition to supporting the methods and operations of mappings, UserDict instances provide the following attribute:

data:

A real dictionary used to store the contents of the UserDict class.

[32]:
```
from collections import UserDict

us = {'name': 'John Doe', 'age': 24}

# Create UserDict object
ud = UserDict(us)
print(ud.data)
```

[32]:
```
{'name': 'John Doe', 'age': 24}
```

Let's say we want to define a custom dictionary object which supports addition operation. When we add two instances of our custom dictionary we want to get a new dictionary with all of the elements in both dictionaries. Keep in mind that, you will get TypeError if you try to add to regular dicts in Python. Let's implement this with the help of UserDict:

[33]:
```
# class for our custom dict
# inherit from UserDict
class AddEnabledDict(UserDict):
    # override the __add__ method
    def __add__(self, other):
        d = AddEnabledDict(self.data)
        d.update(other.data)
        return d

# create custom objects
d_1 = AddEnabledDict(x = 10)
```

```
d_2 = AddEnabledDict(y = 20)
total = d_1 + d_2
print(total)
```

[33]:
```
{'x': 10, 'y': 20}
```

UserList

The **UserList** class acts as a wrapper around list objects. It is a useful base class for your own list-like classes which can inherit from them and override existing methods or add new ones. In this way, one can add new behaviors to lists in Python.

The need for this class has been partially supplanted by the ability to subclass directly from list; however, this class can be easier to work with because the underlying list is accessible as an attribute.

collections.UserList([list]):

Class that simulates a list. The instance's contents are kept in a regular list, which is accessible via the **data** attribute of UserList instances. The instance's contents are initially set to a copy of list, defaulting to the empty list []. **list** parameter can be any iterable, for example a real Python list or a UserList object.

In addition to supporting the methods and operations of mutable sequences, UserList instances provide the following attribute:

data:

A real list object used to store the contents of the UserList class.

Let's say we want to define a list which doesn't allow deleting the items in it. We can easily define such a class by inheriting UserList:

[34]:
```
from collections import UserList

# define a custom class
# this class will inherit from UserList
# it will not allow its items to be deleted
```

```python
# List class in Python has to methods for delete:
# remove() and pop()
class ListWithNoItemDelete(UserList):
    # override remove() method
    def remove(self, s=None):
        self.not_allowed()

    # override pop() method
    def pop(self, s=None):
        self.not_allowed()

    def not_allowed(self):
        raise RuntimeError(“Deletion not allowed”)

# custom list object
custom_list = ListWithNoItemDelete([‘a’, ‘b’, ‘c’])

# try to delete an item
custom_list.remove(‘b’)
```

[34]:
```
RuntimeError: Deletion not allowed
```

UserString

The class, **UserString** acts as a wrapper around string objects. The need for this class has been partially supplanted by the ability to subclass directly from str; however, this class can be easier to work with because the underlying string is accessible as an attribute.

collections.UserString(seq):

Class that simulates a string object. The instance's content is kept in a regular string object, which is accessible via the data attribute of UserString instances. The instance's contents are initially set to a copy of seq. The seq argument can be any object which can be converted into a string using the built-in str() function.

In addition to supporting the methods and operations of strings, UserString instances provide the following attribute:

data:

A real str object used to store the contents of the UserString class.

Let's say we want to define a custom str class that have **concatenate()** method in it:

[35]:
```python
# UserString

from collections import UserString

# define a custom class
# this class will inherit from UserString
class CustomStrClass(UserString):
    # define a new method
    def concatenate(self, other=None, delimiter=' '):
        self.data += delimiter + other

# custom string object
custom_str = CustomStrClass('My Custom')
custom_str.concatenate('String Class')
print(custom_str)
```

[35]:
```
My Custom String Class
```

3. Iterators

Iterables and Iterators

Iterator:

An **Iterator** is an object representing a stream of data and can be iterated upon. In technical terms, a Python iterator is an object that implements the iterator protocol, which consist of two special methods: **__iter__()** and **__next__()**.

Iterable:

An **Iterable** is an object capable of returning its members one at a time. Examples of iterables include all sequence types (such as list, str, and tuple) and some non-sequence types like dict, and file objects. Technically, iterables are objects of any classes with an **__iter__()** method or with a **__getitem__()** method.

iter():

The **iter()** function (which calls the **__iter__()** method behind the scenes) returns an iterator object. So we can say that; an **iterable** is an object which returns an **iterator**.

In this chapter, we will learn how iterators work in Python and how we can define our own iterator classes. You can find the PyCharm project for this chapter in the Github Repository of this book.

Chapter Outline:

- The Iterator Protocol
- Looping Through an Iterator
- Define a Custom Iterator
- Infinite Iterators
- Benefits of Iterators

The Iterator Protocol

In Python, Iterator objects are required to support the following two methods, which together form the **Iterator Protocol**:

`__iter__()`:

Return the iterator object itself. This is required to allow both containers and iterators to be used with the for and in statements. You can use built-in **`iter()`** function which calls the **`__iter__()`** method.

`__next__()`:

Return the next item from the iterator. If there are no further items, raise the **`StopIteration`** exception. You can use built-in **`next()`** function which calls the **`__next__()`** method.

As we learned in the previous section, lists, tuples, dictionaries, and sets are all **iterable** types. In other words, they are the types which you can get an **iterator** from. Let's see some examples:

[1]:
```
# define a tuple (iterable)
tup = ("A", "B", "C")
# get an iterator from the iterable -> iter()
tup_iter = iter(tup)

```

```
6    # call the next() function
7    item_1 = next(tup_iter)
8    print(item_1)
9
10   # call the next() function
11   item_2 = next(tup_iter)
12   print(item_2)
13
14   # call the next() function
15   item_3 = next(tup_iter)
16   print(item_3)
```

[1]:

```
A
B
C
```

In cell 1, we define a tuple which is an iterable. Then in line 4, we call the **iter()** function on this iterable. The **iter()** function returns an iterator and we name it as **tup_iter**. In lines 7 to 16 we call the **next()** function several times. Each time the **next()** function executes, it returns the next item in the iterator.

[2]:

```
1    # define a string (iterable)
2    pyt = 'python'
3    # get an iterator from the iterable -> iter()
4    pyt_iter = pyt.__iter__()
5
```

```
6   # call the next() function
7   item_1 = pyt_iter.__next__()
8   print(item_1)
9
10  # call the next() function
11  item_2 = pyt_iter.__next__()
12  print(item_2)
```

[2]:

```
p
y
```

In cell 2, we call the __ **iter__()** method on a string object. Strings are iterable objects that contain a sequence of characters. The __ **iter__()** method returns an iterator in line 4. And we print the elements in this iterator one by one by calling the __ **next__()** method.

Looping Through an Iterator

As we see in the previous sections, we use the **next()** function (or **__next__()** method) to manually iterate over the items of an iterator. When the **next()** function reaches the end of the iterator, then there is no more data to be returned and you will get an **StopIteration** exception.

[3]:

```
# define a list (iterable)
a_list = [10, 20, 30]

```

```
# get an iterator from the iterable -> iter()
list_iter = iter(a_list)

# call the next() function
print(next(list_iter))
print(next(list_iter))
print(next(list_iter))

# the following next() call will raise exception
print(next(list_iter))
```

[3]:
```
10
20
30
StopIteration
```

In cell 3, we call the **next()** function four times which is more than the number of items in the iterator. And in the last call we get **StopIteration** exception.

The for loop in Python, is able to iterate automatically through any object that can return an iterator. In other words, the for loop can iterate over any iterable object in Python.

[4]:
```
# for loop
for element in a_list:
    print(element)
```

[4]:
```
10
```

```
20

30
```

In cell 4, we use the for loop to iterate over the list we defined in cell 3. As you see, we do not use the **next()** function manually or we don't get any **StopIteration** exception. That's the beauty of the for loop in Python. It handles all of these for us behind the scenes.

Now let's define our own version of the for loop. We will use the while loop and copy the behavior of the for loop. At this point, we have everything we need for this implementation. Let's do it:

[5]:
```python
# custom for loop implementation
# iterable
my_list = [1, 2, 3]

# iterator from this iterable
list_iter = my_list.__iter__()

# while loop
while True:
    try:
        # next item
        element = list_iter.__next__()
        print(element)
    except StopIteration:
        break
```

[5]:

```
1
2
3
```

In cell 5, we implement our own version of the for loop. We use an infinite while loop as: **while True**. We set a try-except block inside the loop. In the try block, we get the next element by calling the **__next__()** method on our iterator. If this call is successful then we print the element. If an error occurs, which is of type **StopIteration**, then we catch that exception in the except block. What we do inside the except block is very simple. We simply break the loop. Which means we have already reached the end of our iterator.

Define a Custom Iterator

Now that we know about iterators and iterator protocol (**__iter__()** and **__next__()** methods) we can define our own iterator classes from scratch.

Defining an iterator is quite easy in Python. All we have to do is to implement **__iter__()** and **__next__()** methods in our class definition. Here are the general rules that we must follow:

- The **__iter__()** method must return the iterator object itself.
- The **__next__()** method must return the next item in the sequence. Also it has to raise a StopIteration exception when it reaches the end of the sequence.

Let's define an iterator object which will generate a sequence of odd numbers like 1, 3, 5, 7, 9, … etc.

[6]:
```python
# custom iterator
class Odd:
    # implement __init__ method
    def __init__(self, limit):
        self.current = 1
        self.limit = limit

    # implement __iter__ method
    # simply return the object itself
    def __iter__(self):
        return self

    # implement __next__ method
    def __next__(self):
        # check if limit reached
        if self.current <= self.limit:
            # get the current value
            current_value = self.current
            # increase the current
            self.current += 2
            return current_value
        # limit is reached so raise exception
        else:
            raise StopIteration
```

In cell 6, we define our own iterator class. It implements both `__iter__()` and `__next__()` methods:

- In the **__iter__()** method, it returns the object itself as: **return self.**
- In the **__next__()** method, it checks if the current value is smaller than or equal to the **limit**. If this is **True**, then it returns the existing value of **self.current** and increases the **self.current** value by **2**. If the **limit** is exceeded than it will raise a **StopIteration** exception.

Now let's call this class and get some odd numbers up to 20:

[7]:
```
# instantiate an Odd object
odd_numbers = Odd(20)

# get first 4 odd numbers
print(odd_numbers.__next__())
print(odd_numbers.__next__())
print(next(odd_numbers))
print(next(odd_numbers))
```

[7]:
```
1
3
5
7
```

In cell 7, we instantiate an object from our **Odd** class. This object will hold the odd numbers from 1 to 20. And we print the first four odd numbers by calling the **next()** method on this object four times.

Since our Odd class is an iterator we can easily set a for loop to iterate over it. Let’s do it:

[8]:
```python
# iterate over Odd class
for n in Odd(8):
    print(n)
```

[8]:
```
1
3
5
7
```

Infinite Iterators

Infinite Iterators are special type objects which has no terminating conditions in their **__next__()** methods. They can be useful when you need to set a counter that you do not know where it will finalize. Let’s define a custom infinite iterator that keeps increasing one by one.

[9]:
```python
class InfiniteCounter:
    def __init__(self):
        self.n = 1

    def __iter__(self):
        return self

    def __next__(self):
        current = self.n
        self.n += 1
        return current
```

Now let’s call our **InfiniteCounter** class and get some numbers in ascending order:

```python
[10]:
# get first four numbers
counter = InfiniteCounter()
print(counter.__next__())
print(counter.__next__())
print(next(counter))
print(next(counter))
```

```
[10]:
1
2
3
4
```

Benefits of Iterators

Use of an iterator simplifies the code and makes it more efficient instead of using a list. For small datasets, iterator and list based approaches have similar performance. But for larger datasets, iterators save both time and memory.

Here are some primary benefits of using iterators:

- Iterators provide cleaner code
- Theoretically, iterators can work with infinite sequences.
- Iterators save resources. Iterator stores only one element in the memory, while list (or tuple) stores all the elements.
- Iterator treats variables of all types, sizes, and shapes uniformly, whether they fit in memory or not.
- Iterator makes recursion unnecessary in handling arrays of arbitrary dimensionality.
- Iterator supports iterating over multiple variables concurrently, because each variable's iteration state is maintained in its own iterator structure.

4. Generators

What is a Generator?

In the previous chapter we learned about Iterators, which are great tools especially when you need to deal with large datasets. However, building an iterator in Python is a bit cumbersome and time consuming. You have to define a new class which implements the iterator protocol (**__iter__()** and **__next__()** methods). In this class, you need to manage internal state of the variables and update them. Moreover you need to raise **StopIteration** exception when there is no value to return back in the **__next__()** method.

Fortunately, we have an elegant solution for this in Python. Python provides Generators to help you easily create iterators. A **Generator** allows you to declare a function that behaves like an iterator, i.e. it can be used in a for loop. In simple terms, a Generator is a function which returns an iterator object. So it's an easy way of creating iterators. You don't need to think about all the work needed when you create an iterator, because the Generator will handle all of them.

In this chapter, we will learn how generators work in Python and how we can define them. You can find the PyCharm project for this chapter in the Github Repository of this book.

Chapter Outline:

- Defining Generators
- Generator Function vs. Normal Function
- Generator Expression
- Benefits of Generators

Defining Generators

As stated in the first section, a generator is a special type of function in Python. This function does not return a single value, instead, it returns an iterator object. In the generator function, we use the **yield** statement instead of the **return** statement. Let's define a simple generator function:

[1]:
```
# define a generator function
def first_generator():
    print('Yielding First item')
    yield 'A'

    print('Yielding Second item')
    yield 'B'

    print('Yielding Last item')
    yield 'C'
```

In cell 1, we define a generator function. The function executes the **yield** statement instead of the **return** keyword. The **yield** statement is what makes this function a generator. When we call this function it will return (yield) an iterator object. Let's see it:

[2]:
```
# call the generator
iter_obj = first_generator()
```

```python

# print first item
first_item = next(iter_obj)
print(first_item)

# print second item
second_item = next(iter_obj)
print(second_item)

# print third item
third_item = next(iter_obj)
print(third_item)
```

[2]:

```
Yielding First item
A
Yielding Second item
B
Yielding Last item
C
```

In cell 2, we call the **`first_generator()`** function which is a generator and returns an iterator object. We name this iterator as **`iter_obj`**. Then we call the **`next()`** function on this iterator object. In each **`next()`** call the iterator executes the **`yield`** statement in respective order and returns an item.

As a rule thumb, the generator function should not include the **`return`** keyword. Because if it includes, then the **`return`** statement will terminate the function.

Now let’s define a more realistic generator by the help of a for loop. In this example we want to define a generator which will keep track of the sequence of numbers starting from zero and up to a given maximum limit.

[3]:
```python
# generator for sequence of numbers
def get_sequence_gen(max):
    for n in range(max):
        yield n

# call the function and get iterator
sequence_iter = get_sequence_gen(10)
# call the next() method
print(sequence_iter.__next__())
print(sequence_iter.__next__())
print(next(sequence_iter))
print(next(sequence_iter))
```

[3]:
```
0
1
2
3
```

In cell 3, we define a generator function which yields the integers from zero up to a given number. As you see, the **yield** statement is inside the for loop. Please be careful that, the value of **n** is stored during successive **next()** calls.

Generator Function vs. Normal Function

A function is a generator function if it contains **at least one yield statement**. It may contain other `yield` or `return` statements if needed. Both `yield` and `return` keywords will return something from a function.

The difference between `return` and `yield` keywords is very crucial for generators. While the `return` statement terminates a function entirely, `yield` statement pauses the function saving all its states and later continues from there on successive calls.

We call the generator function in the same way we call a normal one. During its execution, the generator pauses when it encounters the `yield` keyword. It sends the current value of the iterator stream to the calling environment and wait for the next call. Meanwhile, it saves the local variables and their states internally.

Below are the key points where a generator function differs from a normal function:

- Generator function returns (yields) an iterator object. You don't need to worry about creating this iterator object explicitly, the `yield` keywords does this for you.
- Generator function must contain at least one `yield` statement. It may include multiple `yield` keywords if needed.
- Generator function implements the iterator protocol (`iter()` and `next()` methods) internally.
- Generator function saves the local variables and their states automatically.
- Generator function pauses execution at the yield keyword and pass the control to the caller.
- Generator function raises the `StopIteration` exception automatically when the iterator stream has no value to return.

Let's consider a simple example to demonstrate the difference between a normal function and a generator function. In this example, we want to calculate the sum of **first n** positive integers. To do this, we will define a function that gives us the list of first n positive numbers. We will implement this function in both ways, a normal function and a generator.

Here is the code for the normal function:

[4]:
```
# import the time module
from time import time

# Normal Function
def first_n_numbers(max):
    n, numbers = 1, []
    while n <= max:
        numbers.append(n)
        n += 1
    return numbers

# call the function
start = time()
first_n_list = first_n_numbers(99999999)
sum_of_first_numbers = sum(first_n_list)
print(sum_of_first_numbers)
# elapsed time
end = time()
print("Elapsed Time in seconds:", end - start)
```

[4]:
```
4999999950000000

Elapsed Time in seconds: 17.859
```

In cell 4, we define a normal function that returns the list of first n positive integers. When we call this function it takes a while to complete execution because the list it creates is huge. It also uses a considerable amount of memory to complete this task.

Now let’s define a generator function for the same operation:

[5]:
```
# Generator Function
def first_n_numbers_gen(max):
    n = 1
    while n <= max:
        yield n
        n += 1

# call the function
start = time()
first_n_list = first_n_numbers_gen(99999999)
sum_of_first_numbers = sum(first_n_list)
print(sum_of_first_numbers)
# elapsed time
end = time()
print(“Elapsed Time in seconds:”, end - start)
```

[5]:
```
4999999950000000
```

```
Elapsed Time in seconds: 15.302
```

As you see in cell 5, the generator finishes the same task in less time and it uses less memory resources. Because the generator yields items one by one instead of returning the complete list.

The main reason for performance improvement (when we use generators) is the lazy generation of values. This on demand value generation, results in lower memory usage. One more advantage of generators is, you do not need to wait until all the elements have been generated before you start to use them.

Generator Expression

There are times that you need simple generators for relatively simple tasks in your code. This is where the Generator Expression comes in. You can easily create simple generators on the fly using generator expressions.

Generator expressions are similar to **lambda** functions in Python. Remember that, lambda's are anonymous functions which let us create one-line functions on the fly. Just like a **lambda** function, a generator expression creates an anonymous generator function.

The syntax of a generator expression looks like a list comprehension. The difference is, we have parentheses instead of square brackets in a generator expression. Let's see an example:

[6]:
```
# define a simple list
nums = [1, 2, 3, 4, 5]

```

```
4   # list comprehension
5   num_cubes = [i**3 for i in nums]
6
7   # generator expression
8   cubes_gen = (i**3 for i in nums)
9
10  # print both objects
11  print(num_cubes)
12  print(cubes_gen)
```

[6]:
```
[1, 8, 27, 64, 125]
<generator object <genexpr> at 0x103059d20>
```

In cell 6, line 8 we define a simple generator with the help of the generator expression. Here is the syntax: **`cubes_gen = (i**3 for i in nums)`**. And you see the generator object in the output. As we already know, to be able to get the items in a generator we either need to call the **`next()`** method explicitly or use a for loop to iterate over the generator. Let's print the items in the **`cubes_gen`** object:

[7]:
```
# loop over generator
for item in cubes_gen:
    print(item)
```

[7]:
```
1
8
27
```

```
64

125
```

Let’s do another example. We will define a generator that converts the letters of a string to uppercase. Then we will call the **next()** method to print first two letters.

[8]:
```
# generator for upper case
text = ‘machine learning’
upper_gen = (l.upper() for l in text)
print(upper_gen.__next__())
print(next(upper_gen))
```

[8]:
```
M

A
```

Benefits of Generators

Benefits are great tools especially when you need to deal with large data in relatively limited memory. Here are some key benefits of using generators in Python:

Memory Efficiency:

Let’s assume, you have a normal function that returns a very large sequence. A list with millions of items, for example. You have to wait for this function to finish all the execution and return you the list as a whole. This is obviously not efficient in terms of time and memory resources. On the other hand, if you use a generator function, it will return you the items one by one, and you will have the chance to continue to execute the next lines of code. You don't need to wait for all of the items in the list to be executed by the function. Because the generator will give you one item at a time.

Lazy Evaluation:

Generators provide the power of **lazy evaluation**. Lazy evaluation is computing a value when it is really needed, not when it is instantiated. Let's assume you have a large dataset to compute; lazy evaluation allows you to start using the data immediately while the whole data set is still being computed. Because you do not need the whole data set if you are using a generator.

Implement and Readability:

Generators are very easy to implement and provide code readability. Remember that, you do not need to worry about the **`__iter__()`** and **`__next__()`** methods if you are using a generator. All you need is a simple **`yield`** statement in your function.

Dealing with Infinite Streams:

Generators are wonderful tools when you need to represent an infinite stream of data. An infinite counter, for example. In theory, you cannot store an infinite stream in the memory. You cannot be sure about how much size you will need to store an infinite stream. This is where a generator really shines, since it produces only one item at a time, it can represent an infinite stream of data. And it doesn't have to store all the stream in the memory.

5. Date And Time

Date and Time in Python

Date and Time objects are very important in programming. In Python we have a dedicated module for date and time operations. The **datetime** module supplies classes for manipulating dates and times. While date and time arithmetic is supported, the focus of the implementation of datetime module is on efficient attribute extraction for output formatting and manipulation.

In this chapter, we will learn how to use datetime module in Python. You can find the PyCharm project for this chapter in the Github Repository of this book.

Chapter Outline:

- Aware and Naive Objects
- Constants and Main Classes
- Determining if an Object is Aware or Naïve
- Timedelta Class
- Date Class
- Datetime Class
- Time Class
- Formatting Date and Time

Aware and Naive Objects

Date and time objects may be categorized as “aware” or “naive” depending on whether or not they include timezone information.

With sufficient knowledge of applicable algorithmic and political time adjustments, such as time zone and daylight saving time information, an **aware** object can locate itself relative to other aware objects. An aware object represents a specific moment in time that is not open to interpretation.

A **naive** object does not contain enough information to unambiguously locate itself relative to other date/time objects. Whether a naive object represents Coordinated Universal Time (UTC), local time, or time in some other timezone is purely up to the program, just like it is up to the program whether a particular number represents meters, miles, or mass. Naive objects are easy to understand and to work with, at the cost of ignoring some aspects of reality.

For applications requiring aware objects, datetime and time objects have an optional time zone information attribute, **tzinfo**, that can be set to an instance of a subclass of the abstract **tzinfo class**. These **tzinfo** objects capture information about the offset from UTC time, the time zone name, and whether daylight saving time is in effect.

Only one concrete **tzinfo** class, the **timezone** class, is supplied by the datetime module. The timezone class can represent simple timezones with fixed offsets from UTC, such as UTC itself or North American EST and EDT timezones. Supporting timezones at deeper levels of detail is up to the application. The rules for time adjustment across the world are more political than rational, change frequently, and there is no standard suitable for every application aside from UTC.

Constants and Main Classes

Constants:

The datetime module exports the following constants:

datetime.MINYEAR: The smallest year number allowed in a date or datetime object. MINYEAR is 1.

datetime.MAXYEAR: The largest year number allowed in a date or datetime object. MAXYEAR is 9999.

Available Types:

Here are the built-in types in datetime module in Python. Keep in mind that, objects of these types are immutable.

class datetime.date: An idealized naive date, assuming the current Gregorian calendar always was, and always will be, in effect. Attributes: year, month, and day.

class datetime.time: An idealized time, independent of any particular day, assuming that every day has exactly 24*60*60 seconds. (There is no notion of "leap seconds" here.) Attributes: hour, minute, second, microsecond, and tzinfo.

class datetime.datetime: A combination of a date and a time. Attributes: year, month, day, hour, minute, second, microsecond, and tzinfo.

class datetime.timedelta: A duration expressing the difference between two date, time, or datetime instances to microsecond resolution.

class datetime.tzinfo: An abstract base class for time zone information objects. These are used by the datetime and time classes to provide a customizable notion of time adjustment (for example, to account for time zone and/or daylight saving time).

class datetime.timezone: A class that implements the tzinfo abstract base class as a fixed offset from the UTC.

Common Properties:

The **date**, **datetime**, **time**, and **timezone** types share these common features:

- Objects of these types are immutable.
- Objects of these types are hashable, meaning that they can be used as dictionary keys.
- Objects of these types support efficient pickling via the pickle module.

Determining if an Object is Aware or Naive

Naive and **aware** concepts are crucial in data and time operations. Here are the general rules:

- Objects of the **date** type are always naive.
- An object of type **time** or **datetime** may be aware or naive.
- A **datetime** object **d** is aware if both of the following hold:
    - **d.tzinfo** is not **None**
    - **d.tzinfo.utcoffset(d)** does not return **None**
    - Otherwise, **d** is naive.
- A **time** object **t** is aware if both of the following hold:
    - **t.tzinfo** is not **None**
    - **t.tzinfo.utcoffset(None)** does not return **None**.
    - Otherwise, **t** is naive.
- The distinction between aware and naive doesn't apply to **timedelta** objects.

Timedelta Class

A **timedelta** object represents a duration, the difference between two dates or times.

class datetime.timedelta(days=0, seconds=0, microseconds=0, milliseconds=0, minutes=0, hours=0, weeks=0):

All arguments are optional and default to 0. Arguments may be integers or floats, and may be positive or negative.

Only *days*, *seconds* and *microseconds* are stored internally. Arguments are converted to those units:

- A millisecond is converted to 1000 microseconds.
- A minute is converted to 60 seconds.
- An hour is converted to 3600 seconds.
- A week is converted to 7 days.

and days, seconds and microseconds are then normalized so that the representation is unique, with

- `0 <= microseconds < 1000000`
- `0 <= seconds < 3600*24` (the number of seconds in one day)
- `-999999999 <= days <= 999999999`

The following example illustrates how any arguments besides days, seconds and microseconds are "merged" and normalized into those three resulting attributes:

[1]:
```
from datetime import timedelta

delta = timedelta(
    days=50,
    seconds=27,
    microseconds=10,
    milliseconds=29000,
    minutes=5,
    hours=8,
    weeks=2
)
# Only days, seconds, and microseconds remain
print("Days:", delta.days)
print("Seconds:", delta.seconds)
print("Microseconds:", delta.microseconds)
```

[1]:

```
Days: 64
Seconds: 29156
Microseconds: 10
```

In the next example let's add days two timedelta objects:

[2]:

```
# Add two timedelta objects
delta1 = timedelta(minutes=10, seconds=50)
delta2 = timedelta(hours=25, seconds=20)
delta_sum = delta1 + delta2
print(delta_sum)
```

[2]:

```
1 day, 1:11:10
```

In cell 2, we add two **timedelta** objects and print the result. Addition (+), Subtraction (-), Multiplication (*), Division (/), Floor Division (//) and Modulo (%) operations are supported by the **timedelta** class in Python.

Date Class

A **date** object represents a date (year, month and day) in an idealized calendar, the current Gregorian calendar indefinitely extended in both directions.

January 1 of year 1 is called day number 1, January 2 of year 1 is called day number 2, and so on. 2

class datetime.date(year, month, day):

All arguments are required. Arguments must be integers, in the following ranges:

- MINYEAR <= year <= MAXYEAR
- 1 <= month <= 12
- 1 <= day <= number of days in the given month and year

If an argument outside those ranges is given, **ValueError** is raised.

Class Attributes:

date.min: The earliest representable date, date(MINYEAR, 1, 1).

date.max: The latest representable date, date(MAXYEAR, 12, 31).

date.resolution: The smallest possible difference between non-equal date objects, timedelta(days=1).

<u>Instance Attributes (read-only)</u>:

`date.year`: Between MINYEAR and MAXYEAR inclusive.

`date.month`: Between 1 and 12 inclusive.

`date.day`: Between 1 and the number of days in the given month of the given year.

[3]:
```
from datetime import date

# instantiate a date object: year, month, day
a_valid_date = date(2021, 3, 26)
print(“This is a valid date:”, a_valid_date)

an_ivalid_date = date(2021, 3, 48)
print(“This is an invalid date:”, an_ivalid_date)
```

[3]:
```
This is a valid date: 2021-03-26

ValueError: day is out of range for month
```

`date.today()`:

Return the current local date. Let’s get the current date:

[4]:
```
# current date
current_date = date.today()
print(“Current date is:”, current_date)
print(“Current year:”, current_date.year)
print(“Current month:”, current_date.month)
print(“Current day:”, current_date.day)
```

[4]:
```
Current date is: 2022-03-25

Current year: 2022

Current month: 3

Current day: 25
```

date.fromtimestamp(timestamp):

Return the local date corresponding to the POSIX timestamp.

POSIX timestamp is the time expressed as the number of seconds that have passed since January 1, 1970. That zero moment, known as the **epoch**, is simply the start of the decade in which the Unix operating system (which first used this time representation) was invented.

The **fromtimestamp()** method may raise **OverflowError**, if the timestamp is out of the range of values supported by the platform C **localtime()** function, and **OSError** on **localtime()** failure. It's common for this to be restricted to years from 1970 through 2038. Note that on non-POSIX systems that include leap seconds in their notion of a timestamp, leap seconds are ignored by **fromtimestamp()**.

[5]:
```
# fromtimestamp: get datetime from timestamp
date_time_from_timestamp = date.fromtimestamp(1527635439)
print("Timestamp to datetime:", date_time_from_timestamp)
```

[5]:
```
Timestamp to datetime: 2018-05-30
```

date.isoformat():

Return a string representing the date in ISO 8601 format, YYYY-MM-DD. It is equivalent to **date.__str__()**.

[6]:
```
# isoformat() -> YYYY-MM-DD
iso_formatted_date = date(2002, 12, 4).isoformat()
print(iso_formatted_date)
```

[6]:
```
2002-12-04
```

date.ctime():

Return a string representing the date:

[7]:
```
# isoformat() -> YYYY-MM-DD
iso_formatted_date = date(2002, 12, 4).isoformat()
print(iso_formatted_date)
```

[7]:
```
Wed Dec  4 00:00:00 2002
```

date.fromisoformat(date_string):

Return a date corresponding to a **date_string** given in the format **YYYY-MM-DD**:

[8]:
```
# fromisoformat
date_from_iso_str = date.fromisoformat('2019-12-04')
print(date_from_iso_str)
```

[8]:
```
2019-12-04
```

date.fromordinal(ordinal):

Return the date corresponding to the proleptic Gregorian ordinal, where January 1 of year 1 has ordinal 1. **ValueError** is raised unless `1 <= ordinal <= date.max.toordinal()`.

For any date **d**, **date.fromordinal(d.toordinal()) == d.**

date.strftime(format):

Return a string representing the date, controlled by an explicit format string. Format codes referring to hours, minutes or seconds will see 0 values. **date.__format__(format)** dunder (double underscore) method also works the same.

[9]:
```
# fromordinal() and strftime()
# 730920th day after 1. 1. 0001
d = date.fromordinal(730920)
print(d)

# Methods related to formatting string output
print(d.isoformat())
print(d.strftime(”%d/%m/%y”))
print(d.strftime(“%A %d. %B %Y”))
```

[9]:
```
2002-03-11
2002-03-11
11/03/02
Monday 11. March 2002
```

date.isocalendar():

Return a named tuple object with three components: `year`, `week` and `weekday`. The ISO calendar is a widely used variant of the Gregorian calendar.

The ISO year consists of 52 or 53 full weeks, and where a week starts on a Monday and ends on a Sunday. The first week of an ISO year is the first (Gregorian) calendar week of a year containing a Thursday. This is called week number 1, and the ISO year of that Thursday is the same as its Gregorian year.

For example, 2004 begins on a Thursday, so the first week of ISO year 2004 begins on Monday, 29 Dec 2003 and ends on Sunday, 4 Jan 2004:

[10]:
```python
# isocalendar()
iso_date_year_end = date(2003, 12, 29).isocalendar()
print(iso_date_year_end)

iso_date_year_start = date(2004, 1, 4).isocalendar()
print(iso_date_year_start)
```

[10]:
```
datetime.IsoCalendarDate(year=2004, week=1,
weekday=1)

datetime.IsoCalendarDate(year=2004, week=1,
weekday=7)
```

date.replace(year=self.year, month=self.month, day=self.day):

Return a date with the same value, except for those parameters given new values by whichever keyword arguments are specified.

[11]:
```
# replace()
d = date(2002, 12, 31)
d_new = d.replace(day=26)
print("d:", d)
print("d_new:", d_new)
```

[11]:
```
d: 2002-12-31
d_new: 2002-12-26
```

date.fromisocalendar(year, week, day):

Return a date corresponding to the ISO calendar date specified by year, week and day. This is the inverse of the function **date.isocalendar()**.

date.toordinal():

Return the proleptic Gregorian ordinal of the date, where January 1 of year 1 has ordinal 1. For any date object d, **date.fromordinal(d.toordinal()) == d**.

date.weekday():

Return the day of the week as an integer, where Monday is 0 and Sunday is 6. For example, **date(2002, 12, 4).weekday() == 2**, a Wednesday.

date.isoweekday():

Return the day of the week as an integer, where Monday is 1 and Sunday is 7. For example, **date(2002, 12, 4).isoweekday() == 3**, a Wednesday.

date.timetuple():

Return a **time.struct_time** such as returned by **time.localtime()**, which we will see later.

Datetime Class

A **datetime** object is a single object containing all the information from a **date** object and a **time** object. Like a **date** object, **datetime** assumes the current Gregorian calendar extended in both directions; like a **time** object, **datetime** assumes there are exactly `3600*24` seconds in every day.

datetime.datetime(year, month, day, hour=0, minute=0, second=0, microsecond=0, tzinfo=None, *, fold=0):

The *year*, *month* and *day* arguments are required. *tzinfo* may be **None**, or an instance of a **tzinfo** subclass. The remaining arguments must be integers in the following ranges:

- MINYEAR <= year <= MAXYEAR,
- 1 <= month <= 12,
- 1 <= day <= number of days in the given month and year,
- 0 <= hour < 24,
- 0 <= minute < 60,
- 0 <= second < 60,
- 0 <= microsecond < 1000000,
- fold in [0, 1].

If an argument outside those ranges is given, **ValueError** is raised.

Let's create a **datetime** object with different parameter sets:

[12]:
```python
from datetime import datetime

# call the constructor
datetime_obj = datetime(2020, 5, 29)
```

```
print(datetime_obj)

# call the constructor with time parameters
datetime_obj_with_time = datetime(2020, 5, 29, 8, 45, 52, 162420)
print(datetime_obj_with_time)
```

[12]:

```
2020-05-29 00:00:00
2020-05-29 08:45:52.162420
```

Now let’s get the year, month, hour, minute, seconds, and timestamp attributes from a **`datetime`** object:

[13]:

```
# get attributes
dt = datetime(2019, 8, 17, 23, 38, 54)
print(“Year:”, dt.year)
print(“Month:”, dt.month)
print(“Day:”, dt.day)
print(“Hour:”, dt.hour)
print(“Minute:”, dt.minute)
print(“Seconds:”, dt.second)
print(“Timestamp:”, dt.timestamp())
```

[13]:

```
Year: 2019
Month: 8
Day: 17
Hour: 23
```

```
Minute: 38

Seconds: 54

Timestamp: 1566074334.0
```

datetime.today():

Return the current local datetime, with **tzinfo None**.

datetime.now(tz=None):

Return the current local date and time. If optional argument **tz** is **None** or not specified, this is like **today()**. If **tz** is not **None**, it must be an instance of a **tzinfo** subclass, and the current date and time are converted to tz’s time zone.

[14]:
```
# now()
current_date_time = datetime.now()
print("Current date & time:", current_date_time)
```

[14]:
```
Current date & time: 2022-03-26 12:51:39.045639
```

Class Attributes:

datetime.min: The earliest representable datetime, datetime(MINYEAR, 1, 1, tzinfo=None).

datetime.max: The latest representable datetime, datetime(MAXYEAR, 12, 31, 23, 59, 59, 999999, tzinfo=None).

datetime.resolution: The smallest possible difference between non-equal datetime objects, timedelta(microseconds=1).

Instance Attributes (read-only):

datetime.year: Between MINYEAR and MAXYEAR inclusive.

datetime.month: Between 1 and 12 inclusive.

datetime.day: Between 1 and the number of days in the given month of the given year.

datetime.hour: In range(24).

datetime.minute: In range(60).

datetime.second: In range(60).

datetime.microsecond: In range(1000000).

datetime.tzinfo: The object passed as the tzinfo argument to the datetime constructor, or None if none was passed.

datetime.fold: In [0, 1]. Used to disambiguate wall times during a repeated interval. (A repeated interval occurs when clocks are rolled back at the end of daylight saving time or when the UTC offset for the current zone is decreased for political reasons.) The value 0 (1) represents the earlier (later) of the two moments with the same wall time representation.

Datetime class has the same methods with **date** class. Here is the list of **datetime** class methods:

astimezone():

Returns the datetime object with timezone (tz) information.

combine():

Combines the date and time objects and return a single datetime object.

ctime():

Returns a string representation of date and time.

date():

Return date object with same year, month and day.

fromisoformat():

Returns a datetime object from the string representation of the date and time.

fromordinal():

Returns a date object from the proleptic Gregorian ordinal, where January 1 of year 1 has ordinal 1. The hour, minute, second, and microsecond are 0.

fromtimestamp():

Returns the local date and time corresponding to the POSIX timestamp.

isocalendar():

Returns a named tuple with three components: year, week and weekday.

isoformat():

Returns a string representing the date and time in ISO 8601 format:

- YYYY-MM-DDTHH:MM:SS.ffffff, if microsecond is not 0
- YYYY-MM-DDTHH:MM:SS, if microsecond is 0

isoweekday():

Returns the day of the week as an integer, where Monday is 1 and Sunday is 7.

replace():

Returns a new datetime with the same attributes, except for those attributes given new values by whichever keyword arguments are specified.

strftime():

Returns a string representing the date and time, controlled by an explicit format string.

strptime():

Returns a datetime object corresponding to date string provided as parameter.

time():

Returns time object with same hour, minute, second, microsecond and fold. tzinfo is None.

timetuple():

Returns an object of type **time.struct_time**.

timetz():

Returns time object with same hour, minute, second, microsecond, fold, and tzinfo attributes.

toordinal():

Returns the proleptic Gregorian ordinal of the date. The same as **self.date().toordinal()**.

tzname():

Returns the name of the timezone if tzinfo is not None. If tzinfo is None, returns None.

utcfromtimestamp():

Returns the UTC datetime corresponding to the POSIX timestamp, with tzinfo None. (The resulting object is naive.)

utcoffset():

Returns the UTC offset if tzinfo is not None. If tzinfo is None, returns None.

utcnow():

Returns the current UTC date and time, with tzinfo None. This is like **now()**, but returns the current UTC date and time, as a naive datetime object. An aware current UTC datetime can be obtained by calling **datetime.now(timezone.utc)**.

weekday():

Returns the day of the week as an integer, where Monday is 0 and Sunday is 6.

Let's see some examples of working with **datetime** objects:

[15]:
```
# Using datetime.combine()
d = date(2005, 7, 14)
t = time(12, 30)
combined_dt = datetime.combine(d, t)
print(combined_dt)
```

[15]:
```
2005-07-14 12:30:00
```

In cell 15, we instantiate two objects with **`date()`** and **`time()`** constructors. Then we combine them by using the **`datetime.combine()`** method.

[16]:
```
# Using datetime.now()
# GMT +1
now = datetime.now()
print(now)

# with timezone info
now_tz = datetime.now(timezone.utc)
print(now_tz)
```

[16]:
```
2022-03-26 14:42:10.279281
2022-03-26 11:42:10.279293+00:00
```

In cell 16, we call the **`datetime.now()`** method with and without timezone information.

[17]:
```
# Using datetime.strptime()
dt = datetime.strptime("21/11/06 16:30", "%d/%m/%y %H:%M")
print(dt)
```

[17]:
```
2006-11-21 16:30:00
```

In cell 17, we create a datetime object by calling the **datetime.strptime(date_string, format)** method. We pass the **date_string** and the **format** parameters.

[18]:
```
# Using datetime.timetuple() to get tuple of all
attributes
tt = dt.timetuple()
for it in tt:
    print(it)
```

[18]:
```
2006    # year
11      # month
21      # day
16      # hour
30      # minute
0       # second
1       # weekday (0 = Monday)
325     # number of days since 1st January
-1      # dst - method tzinfo.dst() returned None
```

In cell 18, we print all of the attributes of the dt object which we defined in cell 17.

[19]:
```
# Date in ISO format
ic = dt.isocalendar()
for it in ic:
    print(it)
```

[19]:
```
2006    # ISO year
47      # ISO week
2       # ISO weekday
```

In cell 19, we print the ISO calendar information of our dt object. We get this data by calling the **isocalendar()** method on the datetime object.

[20]:
```
# Formatting a datetime
formatted_dt = dt.strftime(“%A, %d. %B %Y %I:%M%p”)
print(formatted_dt)

dt_str = ‘The {1} is {0:%d}, the {2} is {0:%B}, the {3} is {0:%I:%M%p}.’.format(dt, “day”, “month”, “time”)
print(dt_str)
```

[20]:
```
Tuesday, 21. November 2006 04:30PM
The day is 21, the month is November, the time is 04:30PM.
```

In cell 20, we print the formatted date in two different ways. The first one uses the **datetime.strftime()** method and the second one is the **str.format()** method.

Time Class

A **time** object represents a (local) time of day, independent of any particular day, and subject to adjustment via a **tzinfo** object.

datetime.time(hour=0, minute=0, second=0, microsecond=0, tzinfo=None, *, fold=0):

All arguments are optional. tzinfo may be None, or an instance of a tzinfo subclass. The remaining arguments must be integers in the following ranges:

- 0 <= hour < 24,
- 0 <= minute < 60,
- 0 <= second < 60,
- 0 <= microsecond < 1000000,
- fold in [0, 1].

If an argument outside those ranges is given, **ValueError** is raised. All default to 0 except **tzinfo**, which defaults to None.

Class Attributes:

time.min: The earliest representable time, time(0, 0, 0, 0).

time.max: The latest representable time, time(23, 59, 59, 999999).

time.resolution: The smallest possible difference between non-equal time objects, timedelta(microseconds=1), although note that arithmetic on time objects is not supported.

Instance Attributes (read-only):

time.hour: In range(24).

time.minute: In range(60).

time.second: In range(60).

time.microsecond: In range(1000000).

time.tzinfo: The object passed as the tzinfo argument to the time constructor, or None if none was passed.

time.fold: In [0, 1]. Used to disambiguate wall times during a repeated interval. (A repeated interval occurs when clocks are rolled back at the end of daylight saving time or when the UTC offset for the current zone is decreased for political reasons.) The value 0 (1) represents the earlier (later) of the two moments with the same wall time representation.

time.fromisoformat(time_string):

Returns a time corresponding to a time_string in one of the formats emitted by **time.isoformat()**.

time.replace():

Returns a time with the same value, except for those attributes given new values by whichever keyword arguments are specified.

time.isoformat():

Returns a string representing the time in ISO 8601 format.

time.__str__():

For a time **t**, **str(t)** is equivalent to **t.isoformat()**.

time.strftime(format):

Returns a string representing the time, controlled by an explicit format string.

time.__format__(format):

Same as **time.strftime()**. This makes it possible to specify a format string for a time object in formatted string literals and when using **str.format()**.

time.utcoffset():

If tzinfo is None, returns None, else returns **self.tzinfo.utcoffset(None)**, and raises an exception if the latter doesn't return None or a timedelta object with magnitude less than one day.

time.dst():

If tzinfo is None, returns None, else returns **self.tzinfo.dst(None)**, and raises an exception if the latter doesn't return None, or a timedelta object with magnitude less than one day.

time.tzname():

If tzinfo is None, returns None, else returns **self.tzinfo.tzname(None)**, or raises an exception if the latter doesn't return None or a string object.

Here are some examples of working with a **time** object:

[21]:
```
from datetime import time, tzinfo, timedelta

# define a custom class
class TZ1(tzinfo):
    def utcoffset(self, dt):
        return timedelta(hours=1)
    def dst(self, dt):
        return timedelta(0)
    def tzname(self,dt):
        return “+01:00”
    def  __repr__(self):
        return f"{self.__class__.__name__}()"
```

In cell 21, we define a custom class, **TZ1**, which inherits from **tzinfo** class. And we override some methods in its parent. Let's use this class for creating a **time** object and print some attributes:

[22]:
```
# create a time object
t = time(12, 10, 30, tzinfo=TZ1())
print("t:",t)
print("isoformat:", t.isoformat())
print("dst:", t.dst())
print("tzname:", t.tzname())
print("strftime:", t.strftime("%H:%M:%S %Z"))
print("format:", 'The {} is {:%H:%M}.'.format("time", t))
```

[22]:
```
t: 12:10:30+01:00
isoformat: 12:10:30+01:00
dst: 0:00:00
tzname: +01:00
strftime: 12:10:30 +01:00
format: The time is 12:10.
```

Formatting Date and Time

We already covered the **strftime()** and **strptime()** methods that exist in the datetime class. These are the main methods which are used for date and time formatting.

Here is the reference of all legal format codes that you can use in these methods:

Directive	**Description**	**Example**
%a	Weekday, short version	Wed
%A	Weekday, full version	Wednesday
%w	Weekday as a number 0-6, 0 is Sunday	2
%d	Day of month 01-31	28
%b	Month name, short version	Dec
%B	Month name, full version	December
%m	Month as a number 01-12	12
%y	Year, short version, without century	18
%Y	Year, full version	2021
%H	Hour 00-23	17
%I	Hour 00-12	5
%p	AM/PM	PM
%M	Minute 00-59	41
%S	Second 00-59	8
%f	Microsecond 000000-999999	548513
%z	UTC offset	100
%Z	Timezone	CST
%j	Day number of year 001-366	365
%U	Week number of year, Sunday as the first day of week, 00-53	52

%W	Week number of year, Monday as the first day of week, 00-53	52
%c	Local version of date and time	Mon Dec 31 17:41:00 2018
%C	Century	20
%x	Local version of date	12/31/18
%X	Local version of time	17:41:00
%%	A % character	%
%G	ISO 8601 year	2018
%u	ISO 8601 weekday (1-7)	1
%V	ISO 8601 weeknumber (01-53)	1

Let’s see sone examples for these formats:

[23]:
```
# Formatting Examples
from datetime import datetime

# current date & time
current = datetime.now()
print(“No formatting”, current)

# Weekday short version
weekday_short = current.strftime(“%a %-m %y”)
print(‘Weekday Short Version:’, weekday_short)

# Weekday
```

```
weekday = current.strftime("%A %m %-Y")
print('Weekday:', weekday)

# Hour with PM
pm = current.strftime("%-I %p %S")
print('Hour with PM:', pm)

# Time
common_time = current.strftime("%H:%M:%S")
print('Common Time Representation:',
common_time)
```

[23]:

```
No formatting 2022-03-26 18:17:54.351495

Weekday Short Version: Sat 3 22

Weekday: Saturday 03 2022

Hour with PM: 6 PM 54

Common Time Representation: 18:17:54
```

6. Decorators

Decorators in Python

Decorators are extremely useful tools in Python. A **decorator** is a function that takes another function as the parameter and extends its functionality without explicitly modifying it. It allows us to modify the behavior of a function or a class without touching its source code.

In other words, a decorator wraps a function in order to extend its behaviors, without permanently modifying it.

In this chapter, we will learn how decorators work in Python. You can find the PyCharm project for this chapter in the Github Repository of this book.

Chapter Outline:

- Functions are First-Class Citizens
- Defining a Decorator
- Decorators with Parameters
- General Decorators
- Decorator Changes the Function Name
- Chaining Decorators
- Class Decorators

Functions are First-Class Citizens

In order to understand how decorators work, we need to revisit some important concepts about functions in Python.

In Python, functions are first-class citizens. By this we, mean:

- Functions can be assigned as regular variables
- Functions can be passed as arguments to other functions
- Functions can return functions
- Functions can have other functions (inner functions) in their function body

Now let’s see some examples of these points on functions:

<u>Example 1</u>: Functions can be assigned as regular variables.

```
[1]:  1  # Example 1:
      2  # assign function to a variable
      3  def say_hi(user_name):
      4      return ‘Hi ‘ + user_name
      5
      6  # assign the function
      7  hi_name = say_hi
      8  print(hi_name(‘Bruce Wayne’))
```

```
[1]:     Hi Bruce Wayne
```

In cell 1, we define a function as **`say_hi`**. Then we assign this function to a local variable named **`hi_name`**. Now this **`hi_name`** variable is a function too. And in line 8, we call the **`hi_name`** as: **`hi_name(‘Bruce Wayne’)`**.

<u>Example 2</u>: Functions can be passed as arguments to other functions.

[2]:
```
1   # Example 2:
2   # Functions can be passed as arguments to other functions.
3   def print_hello(user):
4       print('Hello', user)
5
6   def hi_with_function(func, user_name):
7       func(user_name)
8
9   # call the function
10  hi_with_function(print_hello, 'Clark Kent')
```

[2]:
```
Hello Clark Kent
```

In cell 2, we define two functions; **`print_hello`** and **`hi_with_function`**. The second one takes a function as an argument: **`hi_with_function(func, user_name)`**. And it calls this function in its function body in line 7 as: **`func(user_name)`**.

<u>Example 3</u>: Functions can return functions.

[3]:
```
# Example 3:
# Functions can return functions
def return_hi_function():
    return say_hi

# call the function
```

```
7   hi = return_hi_function()
8   print(hi(‘Spiderman’))
```

```
[3]:   Hi Spiderman
```

In cell 3, we define a function named **`return_hi_function`**. This function simply returns another function, which is the **`say_hi`** function that we defined in cell 1. In line 7, we assign the returning function to a variable called **`hi`**. Now this **`hi`** variable is also a function. Then we call it in line 8.

<u>Example 4</u>: Functions can have other functions (inner functions) in their function body.

```
[4]:
# Example 4:
# Functions can have other functions in their body
def outer_func(msg):
    “““Outer function”””

    # define a nested function
    def inner_func():
        “““Inner function”””
        print(msg, ‘from nested function.’)

    # call the nested function
    inner_func()

# call the outer function
```

```
15    outer_func('The Batman')
```

```
[4]:    The Batman from nested function.
```

In cell 4 we define an outer function as **outer_func**. Inside this function we define a nested function named **inner_func**. And in line 12, we call the inner function.

When we call the outer the function in line 15, we pass the text of '**The Batman**' for the value of the **msg** parameter. And the output is **'The Batman from nested function.'**. This text is printed by the **inner_func**. But be careful here, the **inner_func** uses the **msg** variable which is not defined in its own body. In other words, it uses a variable which belongs to its parent's scope. This is the idea behind the **Closures** in Python.

Python Closures: A Closure is a function object that remembers the values in the parent's scope even if they are not present in memory.

Defining a Decorator

A decorator takes in a function as an argument, adds some functionality to it and returns it back to the caller. Sometimes, people call this action as **metaprogramming** because a part of the program tries to modify another part of the program at compile time.

```
[5]:  1    # define a simple decorator
      2    def first_decorator(func):
      3        def wrapper():
```

```
4           print(“Before running {0} function”.format(func.__name__))
5           func()
6           print(“After running {0} function”.format(func.__name__))
7       return wrapper
8
9   def greet():
10      print(“Hi there”)
11
12  # call the first_decorator function
13  greet = first_decorator(greet)
14
15  # call the greet function now
16  greet()
```

[5]:

```
Before running greet function
Hi there
After running greet function
```

In cell 5, we define a simple decorator function. The decorator’s name is **first_decorator** and it has a nested function in it. The nested function is called **wrapper**. The **first_decorator** function simply returns this **wrapper** function.

The **wrapper** function prints some text then calls the function which is the parameter value (func) as: **func()**. And finally, it prints another text.

In line 13, we call the **first_decorator** function by passing the greet function as the argument. And we re-assign the returning value to the **greet** function. Here is the code: **greet =**

first_decorator(greet). Remember that **first_decorator** returns a function (**wrapper**).

In line 16, we call the **greet()** function. And in the output you see that the behavior of **greet()** function is modified. In its original form, it was printing just one line. But now, it prints three lines.

To be sure about the final form of the **greet()** function, let's print its object data:

[6]:
```
# greet function object data
print(greet)
```

[6]:
```
<function first_decorator.<locals>.wrapper at
0x100f569e0>
```

In cell 6, we print the greet function object data. And as you see in the output, the function name is now **wrapper** in the memory. Why? Because we re-assign it from the returning value from the **first_decorator** function, which is the **wrapper** function.

Now we are sure that, we have decorated our **greet()** function. In other words, we didn't modify its source code, but we added some new functionality.

<u>**Decorator Syntax**</u>:

Line 13 in cell 5 is the way we decorate the **greet()** function. It is: **greet = first_decorator(greet)**. What we do is, we pass the function to the decorator and re-assign the resulting value to the same function. In Python, we have a better and more readable syntax for decorating functions. Here it is:

[7]:
```
# Decorator Syntax:
```

```
2
3   # Long Way
4   # def greet():
5   #     print(“Hi there”)
6
7   # greet = first_decorator(greet)
8   # greet()
9
10  # Pythonic Way
11  @first_decorator
12  def greet():
13      print(“Hi there”)
14
15  greet()
```

[7]:

```
Before running greet function
Hi there
After running greet function
```

In cell 7 line 11, you see the syntax for using a decorator. We simply put the decorator’s name with an @ symbol on the function definition line. Here it is:

[8]:

```
@first_decorator
def greet():
    …
```

So, saying **`@first_decorator`** is the simple way of saying **`greet = first_decorator(greet)`**. This is how we apply a decorator to a function.

Decorators with Parameters

In the previous section, we learned how we define and use a decorator function. Now let's see what happens if the function which we want to decorate accepts some parameters. How will we decorate such functions?

Let's assume we want to define a function which takes two numbers, performs the division operation and returns the result. Here is the function:

[9]:
```python
# division function
def division(x, y):
    return x / y

# call the function
result_1 = division(20, 5)
print(result_1)

result_2 = division(8, 0)
print(result_2)
```

[9]:
```
4.0
ZeroDivisionError: division by zero
```

In cell 9, you see the definition of the **division()** function. It simply returns the division result. And we call it with two parameters **(20, 5)** and it returns **4.0**. But there is a problem here. What if the second number, the divisor, is 0? We will get a **ZeroDivisionError** if the second parameter is zero. We can implement a try-except block inside our division function to overcome this problem. But we don't want to modify the code in the function body. So, we need another way.

The solution is to decorate this function with a decorator. The decorator will be responsible of checking if the second parameter is zero or not. Let's define this decorator function:

[10]:
```
# decorator
def division_decorator(f):
    # define the wrapper function
    def wrapper(a, b):
        if b == 0:
            print("Division by zero is not possible.")
            return
        else:
            return f(a, b)

    # return the wrapper function
    return wrapper
```

In cell 10, we define a decorator. The name is **division_decorator** and it is responsible to check for the **ZeroDivisionError**. In the wrapper function, it checks if **b** is equal to zero or not. If it is, then it simply prints an error message and returns. If **b** is not zero, then it calls the function **f** and returns it: **return f(a, b)**. Finally, the **division_decorator** function returns the **wrapper** function. Remember that this is the idea behind the decorators in Python.

Now let's decorate the division function with the **division_decorator**:

[11]:
```
# decorate division function
@division_decorator
def division(x, y):
    return x / y
```

Now that we use the decorator for our **division** function let's call it with zero for the divisor value:

[12]:
```
# call the function now
result_1 = division(20, 5)
print(result_1)

result_2 = division(8, 0)
print(result_2)
```

[12]:
```
4.0
Division by zero is not possible.
None
```

As you see in the output of cell 12, we handle the case where the caller may pass a zero for the divisor. And we managed this by the help of a decorator. The **division_decorator** function implements all the logic which is necessary for this case. More importantly, we didn't modify our **division** function. It’s still the same function, but decorated now.

General Decorators

In the previous section we saw that the parameters of the division function and the wrapper function inside the decorator must match. Why? Because we wrapper function will replace the division function after decoration. So, their parameters should match. But this brings another problem. What if we want to use the same decorator with multiple functions? And what if these functions have different numbers of parameters? Let’s answer this question now.

To start with, let’s assume we want to print our users’ names in full capital letters. Some users may have just their first names, while the others may have first names and last names. So, we will define two separate functions as **first_name** and **full_name**. And we will define a decorator function which will convert the names into upper case. Here is the decorator:

[13]:
```python
# define a general decorator
def upper_decorator(func):
    # wrapper function
    def wrapper(*args):
        # modify the items in *args
        new_args = []
        for i, arg in enumerate(args):
            new_args.append(arg.upper())
        new_args = tuple(new_args)
```

```
10
11          # return the call to the func
12          return func(*new_args)
13
14      # return wrapper function
15      return wrapper
```

In cell 13, we define a general decorator. Why is it general? Because, in its **wrapper** function, the parameter is ***args**. This enables the **wrapper** function to take any number of parameters. In other words, the **wrapper** function will be able to present any function which is decorated with this decorator.

In the **wrapper** function, it modifies the items in the **args** tuple. It converts each item to the upper case and appends it to a list. In line 9, it converts this list to a tuple. And finally, in line 12, it returns the call to the **func** parameter by passing ***new_args** tuple. Here is the line: return **func(*new_args)**.

Now let's define the **first_name** and **full_name** functions. We want both of them to be decorated with the **upper_decorator**. Here they are:

[14]:
```
@upper_decorator
def first_name(name):
    print(name)

# call first_name function
first_name('john')

@upper_decorator
def full_name(firt, last):
    print(firt, last)

# call the full_name function
full_name('john', 'doe')
```

[14]:

```
JOHN
JOHN DOE
```

In cell 14, we define two functions, **first_name** and **full_name**, and we decorate both with the **upper_decorator**. Be careful that, the functions have different number of parameters. Then we call both functions with parameters. And in the output, you see that the names have been capitalized for both. That's the way we use the same generator for functions with any number of parameters.

Here is a more general decorator syntax with both ***args** and ****kwargs**:

[15]:

```
# generator with *args and **kwargs
def most_general_decorator(func):
    def wrapper(*args, **kwargs):
        # implement some logic here
        return func(*args, **kwargs)

    return wrapper
```

Decorator Changes the Function Name

There is an important point which you should always keep in mind when you work with decorators. The function name will be changed after you decorate it. How is that possible? Let’s see with an example:

[16]:
```
1   # function without decorator
2   def last_name(last):
3       print(last)
4
5   # print function name
6   print(“Function name before decorator:”, last_name.__name__)
7
8   # function with decoratoe
9   @upper_decorator
10  def last_name(last):
11      print(last)
12
13  # print function name
14  print(“Function name after decorator:”, last_name.__name__)
```

[16]:
```
Function name before decorator: last_name
Function name after decorator: wrapper
```

In cell 16, we use the upper_decorator which we defined in cell 13. We define a new function as **`last_name`**.

In the first definition, we didn’t decorate it. And we print its name in line 6. The function name is ‘last_name’ as expected.

Then we redefine this function with a decorator this time. And in line 14, we print its name one more time. Surprisingly this time its name is ‘wrapper’.

Why does the name changes from ‘last_name’ to ‘wrapper’? Because when we decorate it, the decorator returns the **wrapper** function. Remember that decorating is exactly the same as this line:

```
last_name = upper_decorator(last_name)
```

We reassign the returning value from the decorator to our function. Since the decorator simply returns the **wrapper** function, the name of our original function changes to ‘wrapper’ now.

To fix this issue, Python provides a very simple solution which is the **functools** module. Let’s see how we can use this module to keep the original function name unchanged:

[17]:

```python
# functools
import functools


# define a decorator with functools
def upper_decorator(func):
    @functools.wraps(func)
    def wrapper(*args):
        # modify the items in *args
        new_args = []
        for i, arg in enumerate(args):
            new_args.append(arg.upper())
        new_args = tuple(new_args)

```

```
14        # return the call to the func
15        return func(*new_args)
16
17    # return wrapper function
18    return wrapper
```

In cell 17 line 6, we use the **functools.wraps()** method. This method is itself a decorator. So, we use it on the **wrapper** function with the **@** symbol. Here is the syntax: **@functools.wraps(func)**. The original function (**func**) is the parameter to this method. This method preserves information about the original function.

Now let's print the name of the **last_name** function one more time:

[18]:

```
# function without decorator
def last_name(last):
    print(last)

# print function name
print("Function name before decorator:", last_name.__name__)

# function with decoratoe
@upper_decorator
def last_name(last):
    print(last)

# print function name
```

```
print("Function name after decorator:",
last_name.__name__)
```

[18]:

```
Function name before decorator: last_name

Function name after decorator: last_name
```

As you see in the output of cell 18, now the name of our original function is the same after we decorate it.

Chaining Decorators

Most of the time you will need to use more than one decorator for a function. This is called chaining decorators on the same function. Let's see how we can use multiple decorators (or the same decorator multiple times).

[19]:

```
# function for printing symbols
def print_symbol(symbol, times):
    print(symbol * times)

# decorator 1
def plus_sign(f):
    def wrapper(*args, **kwargs):
        print_symbol('+', 20)
        f(*args, **kwargs)
        print_symbol('+', 20)
    return wrapper
```

```python

# decorator 2
def minus_sign(f):
    def wrapper(*args, **kwargs):
        print_symbol('-', 20)
        f(*args, **kwargs)
        print_symbol('-', 20)
    return wrapper
```

In cell 19, we define two decorators. Each one prints a different symbol before and after calling the original function. Now let's use both of them on the same function:

[20]:
```python
# chain decorators
@plus_sign
@minus_sign
def say_hi(msg):
    print(msg)

# call the decorated function
say_hi("Hi Python Developer")
```

[20]:
```
++++++++++++++++++++
————————
Hi Python Developer
————————
++++++++++++++++++++
```

In cell 20, we chain two decorators on the **say_hi()** function. And in the output, you see the order of execution of the decorators. The one which is more close to the function definition executes before the others.

Now let's reverse the order of decorators and see the output one more time:

[21]:
```python
# change the order of chaining
@minus_sign
@plus_sign
def say_hi(msg):
    print(msg)
```

[21]:
```
———————
++++++++++++++++++++
Hi Python Developer
++++++++++++++++++++
———————
```

As you see in the output of cell 21, the order of execution changes when we change the order of decorators.

Class Decorators

Decorators can be either functions or classes in Python. In the previous sections we worked with function decorators. Now, we will learn how to define class decorators.

We will define custom classes that acts as a decorator. When a function is decorated with a class, that function becomes an instance of the class. Let’s see how:

```
[22]:
# define a class decorator
class ClassDecorator:
    # init method takes the function
    def __init__(self, func):
        self.func = func

    # implement __call__ method
    def __call__(self):
        # some logic before func call
        print(‘__call__ method before func’)
        self.func()
        # Some logic after func call
        print(‘__call__ method after func’)
```

In cell 22, we have a simple class decorator. For any class to be a decorator, it needs to implement the **__call__()** method. The **__call__()** method acts the same way as the **wrapper** function in the function decorators.

Now let’s use this class to decorate a function:

```
[23]:
# add class decorator to func
@ClassDecorator
def say_hi():
    print(“Hi Python”)
```

```

# call the decorated function
say_hi()
```

[23]:

```
__call__ method before func
Hi Python
__call__ method after func
```

Class Decorator with `*args` and `**kwargs` :

In order to use a class decorator with **`*args`** and **`**kwargs`** arguments, we need to implement the **`__call__()`** method with these arguments and pass them to the decorated function.

[24]:

```
1  # class decorator with *args & **kwargs
2  class ClassDecorator:
3      def __init__(self, func):
4          self.func = func
5  
6      def __call__(self, *args, **kwargs):
7          # some logic before func call
8          self.func(*args, **kwargs)
9          # Some logic after func call
```

In cell 24, the __call__() method of the class decorator takes **`*args`** and **`**kwargs`** arguments. And in line 8, it passes them to the decorated function as: **`self.func(*args, **kwargs)`**.

Let’s decorate a function with this class decorator now:

[25]:
```
# add class decorator to func
@ClassDecorator
def say_hi(first, last, msg='Hi'):
    print("{0} {1} {2}".format(msg, first, last))

# call the decorated function
say_hi("Bruce", "Wayne", "Hi")
```

[25]:
```
Hi Bruce Wayne
```

Class Decorator with the `return` statement:

Remember that, in the **`wrapper`** function of a function decorator we use the **`return`** keyword to return the decorated function. We will do the same thing here, but inside the **`__call__`** method this time.

[26]:
```
# define a class decorator
class UpperDecorator:
    def __init__(self, func):
        self.func = func

    def __call__(self, *args):
        # modify the items in *args
        new_args = []
        for i, arg in enumerate(args):
            new_args.append(arg.upper())
        new_args = tuple(new_args)

```

```
13          # return the call to the func
14          return self.func(*new_args)
15
16  @UpperDecorator
17  def full_name(first, last):
18      print(first, last)
19
20  # call decorated function
21  full_name('jane', 'doe')
```

[26]:

```
JANE DOE
```

In cell 26 line 14, in the **__call__** method we return the decorated function as: **return self.func(*new_args)**.

7. Context Managers

What is a Context Manager?

A **`Context Manager`** is an object that defines the runtime context to be established when executing a **`with`** statement. The context manager handles the entry into, and the exit from, the desired runtime context for the execution of the block of code. Context managers are normally invoked using the **`with`** statement, but can also be used by directly invoking their methods.

Typical uses of context managers include saving and restoring various kinds of global state, locking and unlocking resources, closing opened files, etc…

In this chapter, we will learn how to work with context managers in Python and how to define custom context managers. You can find the PyCharm project for this chapter in the Github Repository of this book.

Chapter Outline:

The **width** Statement

The **`with`** statement is used to wrap the execution of a block with methods defined by a context manager. This allows common **`try…except…finally`** usage patterns to be encapsulated for convenient reuse. Compared to traditional **`try…except…finally`** blocks, the **`with`** statement provide a shorter and reusable code.

In Python Standard Library many classes support **with** statement. A very common example is the built-in **open()** function which provides tools to work with the file objects using the **with** statement.

Here is the general syntax of the **with** statement:

[1]:
```
with expression as target:
    # do something
    # using the target
```

Let's see an example with the **open()** function. We have a text file in the **files** folder in our current project. The file name is **color_names.txt** and it includes some color names. We want to open and print the content in this file by using the **open()** function and **with** statement. Here is the code.

[2]:
```
# define the file path
path = 'files/color_names.txt'

# with statement
with open(path, mode='r') as file:
    # read the file content
    print(file.read())
```

[2]:
```
red
orange
yellow
green
blue
white
```

```
black
```

In cell 2, you see a common use case of the with statement. We use the **open()** function to open the file at the given **path**. And the **open()** function returns the **file** object in read mode. We use this **file** object in line 7 to read and print its content as: **print(file.read())**.

Context Manager Protocol

Python's **with** statement supports the concept of a runtime context defined by a context manager. This is implemented using a pair of methods that allow user-defined classes to define a runtime context that is entered before the statement body is executed and exited when the statement ends.

These methods are called the **Context Manager Protocol**. Here they are:

__enter__(self):

This method is called by the with statement to enter the runtime context related to current object. The with statement will bind this method's return value to the **target** specified in the **as** clause of the statement, if any. See cell 1.

An example of a context manager that returns itself is a **file** object. **File** objects return themselves from **__enter__()** to allow **open()** to be used as the context expression in a **with** statement. See cell 2.

__exit__(self, exc_type, exc_value, traceback):

This method is called when the execution leaves the **with** code block. It exits the runtime context related to this object. The parameters describe the exception that caused the context to be exited. If the context was exited without an exception, all three arguments will be **None**.

If an exception is supplied, and the method wishes to suppress the exception (i.e., prevent it from being propagated), it should return a true value. Otherwise, the exception will be processed normally upon exit from this method.

The `__exit__()` method returns a Boolean value, either `True` or `False`.

The execution of the `with` statement with the methods in context manager protocol proceeds as follows:

[3]:
```
with EXPRESSION as TARGET:
    SUITE
```

1. The context `expression` is evaluated to obtain a context manager.
2. The context manager's `__enter__()` is loaded for later use.
3. The context manager's `__exit__()` is loaded for later use.
4. The context manager's `__enter__()` method is invoked.
5. If a `target` was included in the `with` statement, the return value from `__enter__()` is assigned to it.
6. The suite (code block in the with statement scope) is executed.
7. The context manager's `__exit__()` method is invoked. If an exception caused the suite to be exited, its type, value, and traceback are passed as arguments to `__exit__()`. Otherwise, three `None` arguments are supplied.

   If the suite was exited for any reason other than an exception, the return value from `__exit__()` is ignored, and execution proceeds at the normal location for the kind of exit that was taken.

Creating a Context Manager in Class Form

Now that we know the basic idea behind the context manager protocol let's implement it in a class. This class will be our context manager and we will use it later with the **with** statement.

[4]:
```python
# custom context manager class
class CustomContexManager:
    # init method -> define variables
    def __init__(self, path, mode):
        self.path = path
        self.mode = mode
        self.file = None

    # __enter__ method -> open the file
    def __enter__(self):
        self.file = open(self.path, self.mode)
        return self.file

    # exit method to close the file
    def __exit__(self, exc_type, exc_value,
exc_traceback):
        self.file.close()
```

Our **CustomContexManager** class implements the necessary methods to become a context manager: **__enter__** and **__exit__**.

In its **__init__** method, it defines three instance variables to store the **path**, **mode** and **file** objects.

In the **__enter__** method it uses the built-in **open()** function to open the file in the specified path. Since the **open()** function returns the file object, we assign it to the **self.file** attribute.

In the **__exit__** method we close the file as: **self.file.close()**. The **__exit__** method accepts three arguments, which are required by context manager protocol.

We can use our custom context manager in a **with** statement now. Let’s do it:

[5]:

```
1  # custom contex manager in with statement
2  file_path = ‘files/color_names.txt’
3
4  with CustomContexManager(path=file_path, mode=‘r’) as file:
5      # print the file content
6      print(file.read())
```

[5]:

```
red
orange
yellow
green
blue
white
black
```

In cell 5, we use our **CustomContexManager** class in the with statement. We read file content and print it.

Here is what happens behind the scenes:

1. The line 4 calls the **__enter__** method of the **CustomContexManager** class.
2. The **__enter__** method opens the file and returns it.
3. We name the opened file simply as **file**.
4. In the suite of **with** statement, we read the file content and print it.
5. The **with** statement calls the **__exit__** method.

6. The __exit__ method closes the file.

Let’s define another context manager class. This time we want to print the list of files in the specified folder.

```
[6]:
# context manager for listing files
import os

class ContentList:
    ”‘Prints the content of a directory”’

    def __init__(self, directory):
        self.directory = directory

    def __enter__(self):
        return os.listdir(self.directory)

    def __exit__(self, exc_type, exc_val, exc_tb):
        if exc_type is not None:
            print(“Error getting directory list.”)
        return True

# print the contents of the project directory
project_directory = ‘.’
with ContentList(project_directory) as directory_list:
    print(directory_list)
```

```
[6]:
[‘ClassContextManager.py’, ‘TheWithStatement.py’,
‘files’, ‘.idea’]
```

In cell 6, we define a new context manager. The name of the class is `ContentList`. Why is it a context manager? Because it implements context manager protocol (`__enter__` and `__exit__` methods).

We take the directory path as the parameter in the class constructor which is __init__ method.

In the `__enter__` method we get the list of the contents in this directory simply by calling the `listdir()` method in the `os` module as: `os.listdir(self.directory)`. And we return this list. Please be careful that, our `__enter__` method returns a list in this context manager.

In the __exit__ method, we check if there exist any errors. The `exc_type`, `exc_val`, `exc_tb` parameter values will not be `None` if there is an error in our context manager. So, we check if `exc_type` is not `None` to print an error text.

In line 20, we use our context manager in the `with` statement. Since it returns a list object, we simple assign the returning value to the `directory_list` variable. And in the body of the `with` statement, we print this list. In the cell output, you see the list of the contents in the project directory. Remember that, '.' means the current directory, which is the project directory in our case.

Creating a Context Manager in Function Form

In the previous section we learned how to define context managers by using the class syntax. However, it is a bit cumbersome and lengthy. You need to implement `__enter__` and `__exit__` methods explicitly and you need to handle possible exceptions. Hopefully there is a better way of creating context managers in Python: **function-based context managers**.

Function-Based Context Managers are special functions which use generators and `contextlib.contextmanager` decorator. The `contextlib.contextmanager` decorator is responsible for implementing the context manager protocol.

Let’s define a context manager function:

```
[7]:
1   from contextlib import contextmanager
2
3   # define the context manager function
4   @contextmanager
5   def function_based_context_manager():
6       print(“Entering the context: __enter__”)
7       yield “This is a function based context manager”
8       print(“Leaving the context: __exit__”)
9
10  # use context manager function in with statement
11  with function_based_context_manager() as yield_text:
12      print(yield_text)
```

```
[7]:
Entering the context: __enter__
This is a function based context manager
Leaving the context: __exit__
```

In cell 7, we define a custom function which acts as a context manager. The **contextmanager** decorator is what turns a regular function into a full-stack context manager. You don’t need to worry about implementing the **__enter__** and **__exit__** functions if you implement **@contextmanager** decorator.

The **yield** statement in line 7, acts as the **return** statement in the **__enter__** method in a class-based context manager. Since we have a **yield** statement, function-based context managers are also generator functions.

Let’s define a new context manager. This time it will open a file in write mode and append some text. Here it is:

```
[8]:  1  # context manager for write operations
      2  from contextlib import contextmanager
      3
      4  @contextmanager
      5  def writer_context_manager(path, mode=‘w’):
      6      file_object = None
      7      try:
      8          file_object = open(path, mode=mode)
      9          yield file_object
     10      finally:
     11          if file_object:
     12              file_object.close()
     13
     14  # context manager in with statement
     15  with writer_context_manager(“function_based_context_managers.txt”) as file:
     16      file.write(“The contextlib.contextmanager decorator \n”
     17                 “is responsible for implementing the\n”
     18                  “context manager protocol.”)
```

In cell 8, we define a function-based context manager. In the try block it tries to open the file in the specified path. If it opens it successfully, then it yields (returns) the **`file_object`**. In the finally block we check if we have a **`file_object`** to close. And we close the **`file_object`** if it is not **`None`**.

In the with statement in line 15, we call our context manager with a file name of **`function_based_context_managers.txt`**. The context manager is open this file in write mode and return the file object which we simply name as **`file`**. And in line 16, we write some text in this file. Remember that **'w'** mode will create an empty file, if such a file doesn't exist.

[1] Hands-On Python Series: https://www.amazon.com/gp/product/B09JM26C3Z