Python Ext Advanced Documentation

DEV

sean

2018 11 12

Contents

Python Extra Advanced

written by sean base on following books

Github | https://github.com/newsteinking/High_pythondocExtAdvanced

CHAPTER 1

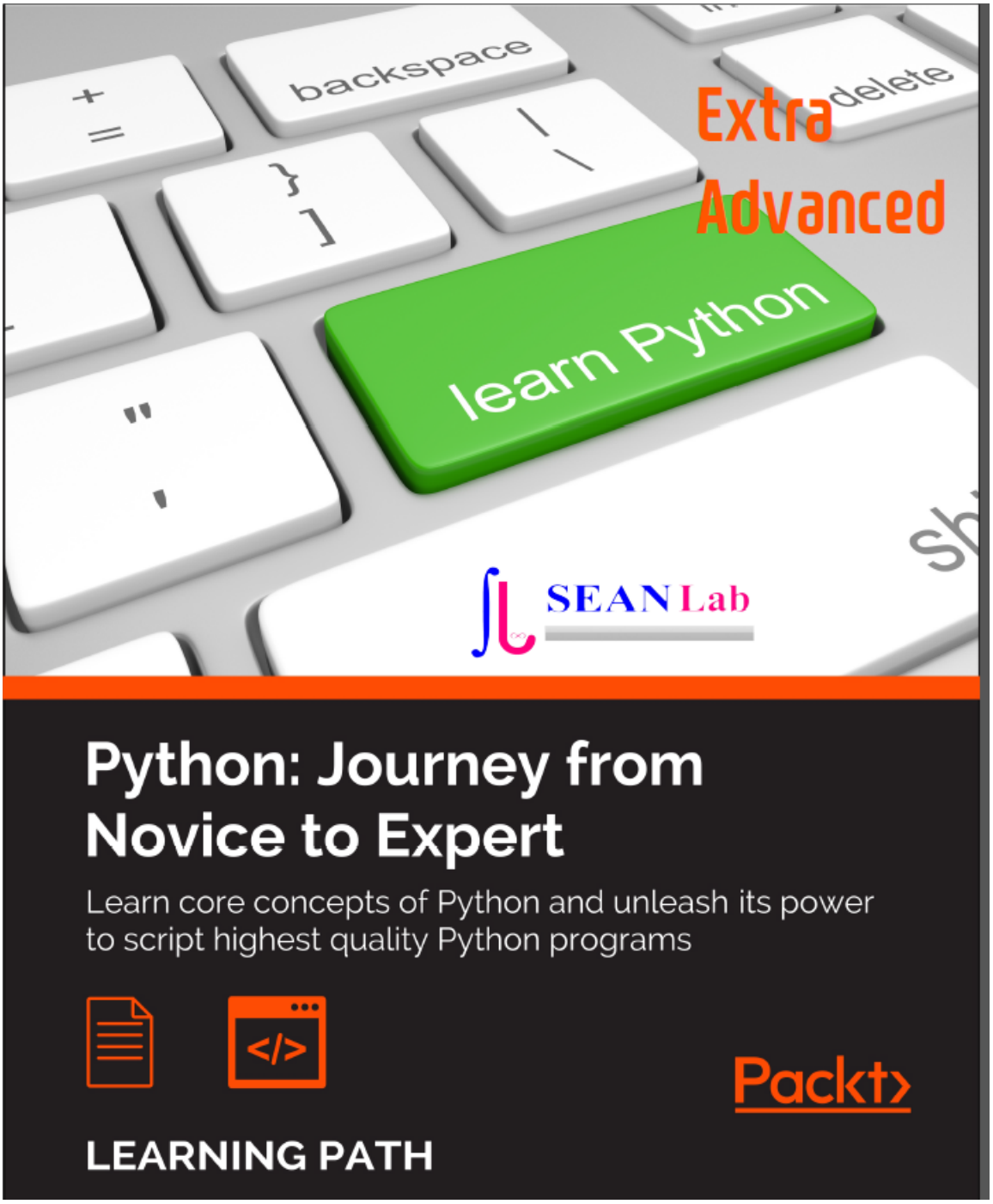
Module 1: Learning Python

1.1 Chapter 0: About

Python Extra Advanced Curriculm

by sean

Base on Python : Journey form Novice to Expert



1.1.1 Thanks to

.

.

- sean

- Mr Ju SS
- OSS Members

1.1.2 SEAN's Paradise

I think that My Life as Software Engineer was torrible , but it's role for social is important so, I keep going for better life & software development

1.2 chapter 1: Introduction

1.2.1 1.1 A proper introduction

1.1.1 Linux

Automatic Install Script

```
$ sudo wget -qO- https://get.docker.com/ | sh
```

remove hell-world

```
$ sudo docker rm `sudo docker ps -aq`
$ sudo docker rmi hello-world
```

.

1.2.2 1.2 Enter the Python

1.2.3 1.3 About Python

1.2.4 1.4 What are the drawbacks?

1.2.5 1.5 Who is using Python today?

1.2.6 1.6 Setting up the environment

1.2.7 1.7 Installing Python

1.2.8 1.8 How you can run a Python program

1.2.9 1.9 How is Python code organized

1.2.10 1.10 Python's execution model

1.2.11 1.11 Guidelines on how to write good code

1.2.12 1.12 The Python culture

1.2.13 1.13 A note on the IDEs

1.2.14 1.14 Summary

1.3 chapter 2: Built-in Data Types

1.3.1 2.1 Everything is an object

2.1.1 Linux

Automatic Install Script

```
$ sudo wget -qO- https://get.docker.com/ | sh
```

remove hell-world

```
$ sudo docker rm `sudo docker ps -aq`
$ sudo docker rmi hello-world
```

.

1.3.2 2.2 Mutable or immutable? That is the question

1.3.3 2.3 Numbers

1.3.4 2.4 Immutable sequences

1.3.5 2.5 Mutable sequences

1.3.6 2.6 Set types

1.3.7 2.7 Mapping types – dictionaries

1.3.8 2.8 The collections module

1.3.9 2.9 Final considerations

1.3.10 2.10 Summary

1.4 chapter 3: Iterating and Making Decisions

1.4.1 3.1 Conditional programming

3.1.1 Linux

Automatic Install Script

```
$ sudo wget -qO- https://get.docker.com/ | sh
```

remove hell-world

```
$ sudo docker rm `sudo docker ps -aq`
$ sudo docker rmi hello-world
```

.

1.4.2 3.2 Looping

1.4.3 3.3 Putting this all together

1.4.4 3.4 A quick peek at the itertools module

1.4.5 3.5 Summary

1.5 chapter 4: Functions, the Building Blocks of Code

1.5.1 4.1 Why use functions?

1.5.2 4.2 Scopes and name resolution

1.5.3 4.3 Input parameters

1.5.4 4.4 Return values

1.5.5 4.5 A few useful tips

1.5.6 4.6 Recursive functions

1.5.7 4.7 Anonymous functions

1.5.8 4.8 Function attributes

```
def multiplication(a, b=1):
"""Return a multiplied by b. """
return a * b


if __name__ == "__main__":

special_attributes = [
    "__doc__", "__name__", "__qualname__", "__module__",
    "__defaults__", "__code__", "__globals__", "__dict__",
    "__closure__", "__annotations__", "__kwdefaults__",
]

for attribute in special_attributes:
    print(attribute, '->', getattr(multiplication, attribute))
```

.

1.5.9 4.9 Built-in functions

. .

```
$any, bin, bool, divmod, filter, float, getattr, id, int, len, list, min, print, set,␣
↪tuple, type, and zip
```

1.5.10 4.10 One final example

. .

. https://docs.python.org/2/library/math.html

math.ceil(x) Return the ceiling of x as a float, the smallest integer value greater than or equal to x.

math.sqrt(x) Return the square root of x.

```
from math import sqrt, ceil


def get_primes(n):
    """Calculate a list of primes up to n (included). """
    primelist = []
    for candidate in range(2, n + 1):
        is_prime = True
        root = int(ceil(sqrt(candidate)))  # division limit
        for prime in primelist:  # we try only the primes
            if prime > root:  # no need to check any further
                break
            if candidate % prime == 0:
                is_prime = False
                break
        if is_prime:
            primelist.append(candidate)
    return primelist


if __name__ == "__main__":

    def test():
        primes = get_primes(10**3)
        primes2 = [
            2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43,
            47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
            107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163,
            167, 173, 179, 181, 191, 193, 197, 199, 211, 223, 227,
            229, 233, 239, 241, 251, 257, 263, 269, 271, 277, 281,
            283, 293, 307, 311, 313, 317, 331, 337, 347, 349, 353,
            359, 367, 373, 379, 383, 389, 397, 401, 409, 419, 421,
            431, 433, 439, 443, 449, 457, 461, 463, 467, 479, 487,
            491, 499, 503, 509, 521, 523, 541, 547, 557, 563, 569,
            571, 577, 587, 593, 599, 601, 607, 613, 617, 619, 631,
            641, 643, 647, 653, 659, 661, 673, 677, 683, 691, 701,
            709, 719, 727, 733, 739, 743, 751, 757, 761, 769, 773,
            787, 797, 809, 811, 821, 823, 827, 829, 839, 853, 857,
            859, 863, 877, 881, 883, 887, 907, 911, 919, 929, 937,
            941, 947, 953, 967, 971, 977, 983, 991, 997
        ]
        return primes == primes2

    print(test())

    print(get_primes(100))
```

.

```
primes = []  # this will contain the primes in the end
upto = 100  # the limit, inclusive
for n in range(2, upto + 1):
    is_prime = True  # flag, new at each iteration of outer for
    for divisor in range(2, n):
        if n % divisor == 0:
            is_prime = False
            break
    if is_prime:  # check on flag
        primes.append(n)

print(primes)
```

1.5.11 4.11 Documenting your code

.

.

: . """ comment """ : , .

```
def square(n):
    """Return the square of a number n. """
    return n ** 2

def get_username(userid):
    """Return the username of a user given their id. """
    return db.get(user_id=userid).username


def connect(host, port, user, password):
    """Connect to a database.

    Connect to a PostgreSQL database directly, using the given
    parameters.

    :param host: The host IP.
    :param port: The desired port.
    :param user: The connection username.
    :param password: The connection password.
    :return: The connection object.
    """
    # body of the function here...
    return connection
```

1.5.12 4.12 Importing objects

. .

```
import module_name
from module_name import function_name
from mymodule import myfunc as better_named_func  ##
from module_name import *    ##    import,
```

lib

```
├── func_from.py
├── func_import.py
├── lib
├── funcdef.py
└── __init__.py
```

__init__.py .

. fundef.py

```
def square(n):
    return n ** 2


def cube(n):
    return n ** 3
```

func_import.py

```
import lib.funcdef


print(lib.funcdef.square(10))
print(lib.funcdef.cube(10))
```

func_from.py

```
from lib.funcdef import square, cube

print(square(10))
print(cube(10))
```

1.5.13 4.13 Ralative import

Absolute Imports

An absolute import specifies the resource to be imported using its full path from the project's root folder.

.

```
└── project
    ├── package1
    │   ├── module1.py
    │   └── module2.py
    └── package2
        ├── __init__.py
        ├── module3.py
        ├── module4.py
        └── subpackage1
            └── module5.py
```

Absolute imports . .. code-block:: python

from package1 import module1 from package1.module2 import function1 from package2 import class1 from package2.subpackage1.module5 import function2

Relative Imports A relative import specifies the resource to be imported relative to the current location—that is, the location where the import statement is

.

```
from .some_module import some_class
from ..some_package import some_function
from . import some_class
```

One clear advantage of relative imports is that they are quite succinct()

1.5.14 4.14 Summary

import .

1.6 chapter 5: Saving Time and Memory

.

1.6.1 5.1 map, zip, and filter

map,zip,filter .

Map

Map

```
map(function, iterable, ...) returns an iterator that applies function
to every item of iterable, yielding the results. If additional iterable arguments are
passed, function must take that many arguments and is applied to the items from
all iterables in parallel. With multiple iterables, the iterator stops when the␣
↪shortest
iterable is exhausted
```

.

```
>>> map(lambda *a: a, range(3))  # without wrapping in list...
<map object at 0x7f563513b518>  # we get the iterator object
>>> list(map(lambda *a: a, range(3)))  # wrapping in list...
[(0,), (1,), (2,)]  # we get a list with its elements
>>> list(map(lambda *a: a, range(3), 'abc'))  # 2 iterables
[(0, 'a'), (1, 'b'), (2, 'c')]
>>> list(map(lambda *a: a, range(3), 'abc', range(4, 7)))  # 3
[(0, 'a', 4), (1, 'b', 5), (2, 'c', 6)]
>>> # map stops at the shortest iterator
>>> list(map(lambda *a: a, (), 'abc'))  # empty tuple is shortest
[]
>>> list(map(lambda *a: a, (1, 2), 'abc'))  # (1, 2) shortest
[(1, 'a'), (2, 'b')]
>>> list(map(lambda *a: a, (1, 2, 3, 4), 'abc'))  # 'abc' shortest
[(1, 'a'), (2, 'b'), (3, 'c')]
```

range(3) lamda map object list . 2,3 .

continue ...

zip

.

```
zip(*iterables) returns an iterator of tuples, where the i-th tuple contains the
i-th element from each of the argument sequences or iterables. The iterator stops
when the shortest input iterable is exhausted. With a single iterable argument, it
returns an iterator of 1-tuples. With no arguments, it returns an empty iterator.
```

zip . .

```
>>> grades = [18, 23, 30, 27, 15, 9, 22]
>>> avgs = [22, 21, 29, 24, 18, 18, 24]
>>> list(zip(avgs, grades))
[(22, 18), (21, 23), (29, 30), (24, 27), (18, 15), (18, 9), (24, 22)]
>>> list(map(lambda *a: a, avgs, grades))  # equivalent to zip
[(22, 18), (21, 23), (29, 30), (24, 27), (18, 15), (18, 9), (24, 22)]
```

zip map .

```
>>> a = [5, 9, 2, 4, 7]
>>> b = [3, 7, 1, 9, 2]
>>> c = [6, 8, 0, 5, 3]
>>> maxs = map(lambda n: max(*n), zip(a, b, c))
>>> list(maxs)
[6, 9, 2, 9, 7]
```

filter

.

```
filter(function, iterable) construct an iterator from those elements
of iterable for which function returns True. iterable may be either a sequence, a
container which supports iteration, or an iterator. If function is None, the identity
function is assumed, that is, all elements of iterable that are false are removed.
```

```
>>> test = [2, 5, 8, 0, 0, 1, 0]
>>> list(filter(None, test))
[2, 5, 8, 1]
>>> list(filter(lambda x: x, test))  # equivalent to previous one
[2, 5, 8, 1]
>>> list(filter(lambda x: x > 4, test))  # keep only items > 4
[5, 8]
```

1.6.2 5.2 Comprehensions

comprehensions list,dict,set .

```
>>> squares = []
>>> for n in range(10):
...     squares.append(n ** 2)
...
>>> list(squares)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

(continues on next page)

( )

```
# This is better, one line, nice and readable
>>> squares = map(lambda n: n**2, range(10))
>>> list(squares)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

.

```
>>> [n ** 2 for n in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Nested Comprehensions

.

```
items = 'ABCDE'
pairs = []

for a in range(len(items)):
    for b in range(a, len(items)):
        pairs.append((items[a], items[b]))

print(pairs)
```

list comprehensions .

```
items = 'ABCDE'
pairs = [(items[a], items[b])
    for a in range(len(items)) for b in range(a, len(items))]

print(pairs)
```

Filtering a comprehension

Pythagorean triple (a2 + b2 = c2)

```
from math import sqrt

# this will generate all possible pairs
mx = 10
legs = [(a, b, sqrt(a**2 + b**2))
    for a in range(1, mx) for b in range(a, mx)]
# this will filter out all non pythagorean triples
legs = list(
    filter(lambda triple: triple[2].is_integer(), legs))

print(legs)  # prints: [(3, 4, 5.0), (6, 8, 10.0)]
```

intger .

```
from math import sqrt

mx = 10
```

(continues on next page)

( )

```
legs = [(a, b, sqrt(a**2 + b**2))
    for a in range(1, mx) for b in range(a, mx)]
legs = filter(lambda triple: triple[2].is_integer(), legs)

# this will make the third number in the tuples integer
legs = list(
    map(lambda triple: triple[:2] + (int(triple[2]), ), legs))

print(legs)  # prints: [(3, 4, 5), (6, 8, 10)]
```

list comprehesion .

```
from math import sqrt
# this step is the same as before
mx = 10
legs = [(a, b, sqrt(a**2 + b**2))
    for a in range(1, mx) for b in range(a, mx)]
# here we combine filter and map in one CLEAN list comprehension
legs = [(a, b, int(c)) for a, b, c in legs if c.is_integer()]

print(legs)  # prints: [(3, 4, 5), (6, 8, 10)]
```

1.6.3 5.3 Generators

1.6.4 5.4 Some performance considerations

1.6.5 5.5 Don't overdo comprehensions and generators

1.6.6 5.6 Name localization

1.6.7 5.7 Generation behavior in built-ins

1.6.8 5.8 One last example

1.6.9 5.9 Summary

1.7 chapter 6: Advanced Concepts

1.7.1 6.1 Decorators

6.1.1 Linux

Automatic Install Script

```
$ sudo wget -qO- https://get.docker.com/ | sh
```

remove hell-world

```
$ sudo docker rm `sudo docker ps -aq`
$ sudo docker rmi hello-world
```

.

1.7.2 6.2 Object-oriented programming

1.7.3 6.3 Writing a custom iterator

1.7.4 6.4 Summary

1.8 chapter 7: Testing, Profiling, and Dealing with Exceptions

1.8.1 7.1 Testing your application

7.1.1 Linux

Automatic Install Script

```
$ sudo wget -qO- https://get.docker.com/ | sh
```

remove hell-world

```
$ sudo docker rm `sudo docker ps -aq`
$ sudo docker rmi hello-world
```

.

1.8.2 7.2 Test-driven development

1.8.3 7.3 Exceptions

1.8.4 7.4 Profiling Python

1.8.5 7.4 Summary

1.9 chapter 8: The Edges – GUIs and Scripts

1.9.1 8.1 First approach

8.1.1 Linux

Automatic Install Script

```
$ sudo wget -qO- https://get.docker.com/ | sh
```

remove hell-world

```
$ sudo docker rm `sudo docker ps -aq`
$ sudo docker rmi hello-world
```

.

1.9.2 8.2 Second approach

1.9.3 8.3 Where do we go from here?

1.9.4 8.4 Summary

1.10 chapter 9: Data Science

1.10.1 9.1 IPython and Jupyter notebook

9.1.1 Linux

Automatic Install Script

```
$ sudo wget -qO- https://get.docker.com/ | sh
```

remove hell-world

```
$ sudo docker rm `sudo docker ps -aq`
$ sudo docker rmi hello-world
```

.

1.10.2 9.2 Dealing with data

1.10.3 9.3 Where do we go from here?

1.10.4 9.4 Summary

1.11 chapter 10: Web Development Done Right

What is the Web? 311 How does the Web work? 312 The Django web framework 313 A regex website 316 The future of web development 338 Summary

1.11.1 10.1 What is the Web?

10.1.1 Linux

Automatic Install Script

```
$ sudo wget -qO- https://get.docker.com/ | sh
```

remove hell-world

```
$ sudo docker rm `sudo docker ps -aq`
$ sudo docker rmi hello-world
```

.

1.11.2 10.2 How does the Web work?

1.11.3 10.3 The Django web framework

1.11.4 10.4 A regex website

1.11.5 10.5 The future of web development

1.11.6 10.6 Summary

1.12 chapter 11: Debugging and Troubleshooting

Debugging techniques 344 Troubleshooting guidelines 357 Summary

1.12.1 11.1 Debugging techniques

11.1.1 Linux

Automatic Install Script

```
$ sudo wget -qO- https://get.docker.com/ | sh
```

remove hell-world

```
$ sudo docker rm `sudo docker ps -aq`
$ sudo docker rmi hello-world
```

.

1.12.2 11.2 Troubleshooting guidelines

1.12.3 11.3 Summary

1.13 chapter 12: Summing Up – A Complete Example

The challenge 359 Our implementation 360 Implementing the Django interface 360 Implementing the Falcon API 387 Where do you go from here? 404 Summary 405 A word of farewell 406

1.13.1 12.1 The challenge

12.1.1 Linux

Automatic Install Script

```
$ sudo wget -qO- https://get.docker.com/ | sh
```

remove hell-world

```
$ sudo docker rm `sudo docker ps -aq`
$ sudo docker rmi hello-world
```

.

1.13.2 12.2 Our implementation

1.13.3 12.3 Implementing the Django interface

1.13.4 12.4 Implementing the Falcon API

1.13.5 12.5 Where do you go from here?

1.13.6 12.6 Summary

1.13.7 12.7 A word of farewell

CHAPTER 2

Module 2: Python 3 Object-Oriented Programming

2.1 chapter 1: Object-oriented Design

Introducing object-oriented 409 Objects and classes 411 Specifying attributes and behaviors 413 Hiding details and creating the public interface 417 Composition 419 Inheritance 422 Case study 426 Exercises 433 Summary

2.1.1 1.1 Introducing object-oriented

1.1.1 Linux

Automatic Install Script

```
$ sudo wget -qO- https://get.docker.com/ | sh
```

remove hell-world

```
$ sudo docker rm `sudo docker ps -aq`
$ sudo docker rmi hello-world
```

.

2.1.2 1.2 Objects and classes

2.1.3 1.3 Specifying attributes and behaviors

2.1.4 1.4 Hiding details and creating the public interface

2.1.5 1.5 Composition

2.1.6 1.6 Inheritance

2.1.7 1.7 Case study

2.1.8 1.8 Exercises

2.1.9 1.9 Summary

2.2 chapter 2: Objects in Python

Creating Python classes 435 Modules and packages 445 Organizing module contents 451 Who can access my data? 454 Third-party libraries 456 Case study 457 Exercises 466 Summary

2.2.1 2.1 Creating Python classes

2.1.1 Linux

Automatic Install Script

```
$ sudo wget -qO- https://get.docker.com/ | sh
```

remove hell-world

```
$ sudo docker rm `sudo docker ps -aq`
$ sudo docker rmi hello-world
```

.

2.2.2 2.2 Modules and packages

2.2.3 2.3 Organizing module contents

2.2.4 2.4 Who can access my data?

2.2.5 2.5 Third-party libraries

2.2.6 2.6 Case study

2.2.7 2.7 Exercises

2.2.8 2.8 Summary

2.3 chapter 3: When Objects Are Alike

Basic inheritance 467 Multiple inheritance 473 Polymorphism 483 Abstract base classes 486 Case study 490 Exercises 503 Summary

2.3.1 3.1 Basic inheritance

3.1.1 Linux

Automatic Install Script

```
$ sudo wget -qO- https://get.docker.com/ | sh
```

remove hell-world

```
$ sudo docker rm `sudo docker ps -aq`
$ sudo docker rmi hello-world
```

.

2.3.2 3.2 Multiple inheritance

2.3.3 3.3 Polymorphism

2.3.4 3.4 Abstract base classes

2.3.5 3.5 Case study

2.3.6 3.6 Exercises

2.3.7 3.7 Summary

2.4 chapter 4: Expecting the Unexpected

Raising exceptions 506 Case study 522 Exercises 531 Summary

2.4.1 4.1 Raising exceptions

4.1.1 Linux

Automatic Install Script

```
$ sudo wget -qO- https://get.docker.com/ | sh
```

remove hell-world

```
$ sudo docker rm `sudo docker ps -aq`
$ sudo docker rmi hello-world
```

.

2.4.2 4.2 Case study

2.4.3 4.3 Exercises

2.4.4 4.4 Summary

2.5 chapter 5: When to Use Object-oriented Programming

Treat objects as objects 533 Adding behavior to class data with properties 537 Manager objects 546 Case study 553 Exercises 561 Summary

2.5.1 5.1 Treat objects as objects

5.1.1 Linux

Automatic Install Script

```
$ sudo wget -qO- https://get.docker.com/ | sh
```

remove hell-world

```
$ sudo docker rm `sudo docker ps -aq`
$ sudo docker rmi hello-world
```

.

2.5.2 5.2 Adding behavior to class data with properties

2.5.3 5.3 Manager objects

2.5.4 5.4 Case study

2.5.5 5.5 Exercises

2.5.6 5.6 Summary

2.6 chapter 6: Python Data Structures

Empty objects 563 Tuples and named tuples 565 Dictionaries 568 Lists 575 Sets 581 Extending built-ins 585 Queues 590 Case study 596 Exercises 602 Summary

2.6.1 6.1 Empty objects

6.1.1 Linux

Automatic Install Script

```
$ sudo wget -qO- https://get.docker.com/ | sh
```

remove hell-world

```
$ sudo docker rm `sudo docker ps -aq`
$ sudo docker rmi hello-world
```

.

2.6.2 6.2 Tuples and named tuples

2.6.3 6.3 Dictionaries

2.6.4 6.4 Lists

2.6.5 6.5 Sets

2.6.6 6.6 Extending built-ins

2.6.7 6.7 Queues

2.6.8 6.8 Case study

2.6.9 6.9 Exercises

2.6.10 6.10 Summary

2.7 chapter 7: Python Object-oriented Shortcuts

Python built-in functions 605 An alternative to method overloading 613 Functions are objects too 621 Case study 627 Exercises 634 Summary

2.7.1 7.1 Python built-in functions

7.1.1 Linux

Automatic Install Script

```
$ sudo wget -qO- https://get.docker.com/ | sh
```

remove hell-world

```
$ sudo docker rm `sudo docker ps -aq`
$ sudo docker rmi hello-world
```

.

2.7.2 7.2 An alternative to method overloading

2.7.3 7.3 Functions are objects too

2.7.4 7.4 Case study

2.7.5 7.5 Exercises

2.7.6 7.6 Summary

2.8 chapter 8: Strings and Serialization

Strings 637 Regular expressions 652 Serializing objects 660 Case study 668 Exercises 673 Summary

2.8.1 8.1 Strings

8.1.1 Linux

Automatic Install Script

```
$ sudo wget -qO- https://get.docker.com/ | sh
```

remove hell-world

```
$ sudo docker rm `sudo docker ps -aq`
$ sudo docker rmi hello-world
```

.

2.8.2 8.2 Regular expressions

2.8.3 8.3 Serializing objects

2.8.4 8.4 Case study

2.8.5 8.5 Exercises

2.8.6 8.6 Summary

2.9 chapter 9: The Iterator Pattern

Design patterns in brief 677 Iterators 678 Comprehensions 681 Generators 687 Coroutines 692 Case study 699 Exercises 706 Summary 707

2.9.1 9.1 Design patterns in brief

9.1.1 Linux

Automatic Install Script

```
$ sudo wget -qO- https://get.docker.com/ | sh
```

remove hell-world

```
$ sudo docker rm `sudo docker ps -aq`
$ sudo docker rmi hello-world
```

.

2.9.2 9.2 Iterators

2.9.3 9.3 Comprehensions

2.9.4 9.4 Generators

2.9.5 9.5 Coroutine

2.9.6 9.6 Case study

2.9.7 9.7 Exercises

2.9.8 9.8 Summary

2.10 chapter 10: Python Design Patterns I

The decorator pattern 709 The observer pattern 715 The strategy pattern 718 The state pattern 721 The singleton pattern 728 The template pattern 733 Exercises 737 Summary

2.10.1 10.1 The decorator pattern

10.1.1 Linux

Automatic Install Script

```
$ sudo wget -qO- https://get.docker.com/ | sh
```

remove hell-world

```
$ sudo docker rm `sudo docker ps -aq`
$ sudo docker rmi hello-world
```

.

2.10.2 10.2 The observer pattern

2.10.3 10.3 The strategy pattern

2.10.4 10.4 The state pattern

2.10.5 10.5 The singleton pattern

2.10.6 10.6 The template pattern

2.10.7 10.7 Exercises

2.10.8 10.8 Summary

2.11 chapter 11: Python Design Patterns II

The adapter pattern 739 The facade pattern 743 The flyweight pattern 745 The command pattern 749 The abstract factory pattern 754 The composite pattern 759 Exercises 763 Summary

2.11.1 11.1 The adapter pattern

11.1.1 Linux

Automatic Install Script

```
$ sudo wget -qO- https://get.docker.com/ | sh
```

remove hell-world

```
$ sudo docker rm `sudo docker ps -aq`
$ sudo docker rmi hello-world
```

.

2.11.2 11.2 The facade pattern

2.11.3 11.3 The flyweight pattern

2.11.4 11.4 The command pattern

2.11.5 11.5 The abstract factory pattern

2.11.6 11.6 The composite pattern

2.11.7 11.7 Exercises

2.11.8 11.8 Summary

2.12 chapter 12: Testing Object-oriented Programs

Why test? 765 Unit testing 768 Testing with py.test 776 Imitating expensive objects 786 How much testing is enough? 790 Case study 793 Exercises 799 Summary

2.12.1 12.1 Why test?

12.1.1 Linux

Automatic Install Script

```
$ sudo wget -qO- https://get.docker.com/ | sh
```

remove hell-world

```
$ sudo docker rm `sudo docker ps -aq`
$ sudo docker rmi hello-world
```

.

2.12.2 12.2 Unit testing

2.12.3 12.3 Testing with py.test

2.12.4 12.4 Imitating expensive objects

2.12.5 12.5 How much testing is enough?

2.12.6 12.6 Case study

2.12.7 12.7 Exercises

2.12.8 12.8 Summary

2.13 chapter 13: Concurrency

Threads 802 Multiprocessing 807 Futures 814 AsyncIO 817 Case study 826 Exercises 833 Summary

2.13.1 13.1 Threads

13.1.1 Linux

Automatic Install Script

```
$ sudo wget -qO- https://get.docker.com/ | sh
```

remove hell-world

```
$ sudo docker rm `sudo docker ps -aq`
$ sudo docker rmi hello-world
```

.

2.13.2 13.2 Multiprocessing

2.13.3 13.3 Futures

2.13.4 13.4 AsyncIO

2.13.5 13.5 Case study

2.13.6 13.6 Exercises

2.13.7 13.7 Summary

CHAPTER 3

Module 3: Mastering Python

3.1 chapter 1: Getting Started – One Environment per Project

Creating a virtual Python environment using venv 838 Bootstrapping pip using ensurepip 843 Installing C/C++ packages 844 Summary

3.1.1 1.1 Creating a virtual Python environment using venv

1.1.1 Linux

Automatic Install Script

```
$ sudo wget -qO- https://get.docker.com/ | sh
```

remove hell-world

```
$ sudo docker rm `sudo docker ps -aq`
$ sudo docker rmi hello-world
```

.

3.1.2 1.2 Bootstrapping pip using ensurepip

3.1.3 1.3 Installing C/C++ packages

3.1.4 1.4 Summary

3.2 chapter 2: Pythonic Syntax, Common Pitfalls, and Style Guide

Code style – or what is Pythonic code? 850 Common pitfalls 871 Summary

3.2.1 2.1 ode style – or what is Pythonic code?

2.1.1 Linux

Automatic Install Script

```
$ sudo wget -qO- https://get.docker.com/ | sh
```

remove hell-world

```
$ sudo docker rm `sudo docker ps -aq`
$ sudo docker rmi hello-world
```

.

3.2.2 2.2 Common pitfalls

3.2.3 2.3 Summary

3.3 chapter 3: Containers and Collections – Storing Data the Right Way

Time complexity – the big O notation 886 Core collections 887 Advanced collections 898 Summary

3.3.1 3.1 Time complexity – the big O notation

3.1.1 Linux

Automatic Install Script

```
$ sudo wget -qO- https://get.docker.com/ | sh
```

remove hell-world

```
$ sudo docker rm `sudo docker ps -aq`
$ sudo docker rmi hello-world
```

.

3.3.2 3.2 Core collections

3.3.3 3.3 Advanced collections

3.3.4 3.4 Summary

3.4 chapter 4: Functional Programming – Readability Versus Brevity

Functional programming 918 list comprehensions 918 dict comprehensions 921 set comprehensions 922 lambda functions 922 functools 925 itertools 931 Summary

3.4.1 4.1 Functional programming

4.1.1 Linux

Automatic Install Script

```
$ sudo wget -qO- https://get.docker.com/ | sh
```

remove hell-world

```
$ sudo docker rm `sudo docker ps -aq`
$ sudo docker rmi hello-world
```

.

3.4.2 4.2 list comprehensions

3.4.3 4.3 dict comprehensions

3.4.4 4.4 set comprehensions

3.4.5 4.5 lambda functions

3.4.6 4.6 functools

3.4.7 4.7 itertools

3.4.8 4.8 Summary

3.5 chapter 5: Decorators – Enabling Code Reuse by Decorating

Decorating functions 940 Decorating class functions 952 Decorating classes 961 Useful decorators 966 Summary

3.5.1 5.1 Decorating functions

5.1.1 Linux

Automatic Install Script

```
$ sudo wget -qO- https://get.docker.com/ | sh
```

remove hell-world

```
$ sudo docker rm `sudo docker ps -aq`
$ sudo docker rmi hello-world
```

.

3.5.2 5.2 Decorating class functions

3.5.3 5.3 Decorating classes

3.5.4 5.4 Useful decorators

3.5.5 5.5 Summary

3.6 chapter 6: Generators and Coroutines – Infinity, One Step at a Time

What are generators? 978 Coroutines 990 Summary

3.6.1 6.1 What are generators?

6.1.1 Linux

Automatic Install Script

```
$ sudo wget -qO- https://get.docker.com/ | sh
```

remove hell-world

```
$ sudo docker rm `sudo docker ps -aq`
$ sudo docker rmi hello-world
```

.

3.6.2 6.2 Coroutines

3.6.3 6.3 Summary

3.7 chapter 7: Async IO – Multithreading without Threads

Introducing the asyncio library 1004 Summary

3.7.1 7.1 Introducing the asyncio library

7.1.1 Linux

Automatic Install Script

```
$ sudo wget -qO- https://get.docker.com/ | sh
```

remove hell-world

```
$ sudo docker rm `sudo docker ps -aq`
$ sudo docker rmi hello-world
```

.

3.7.2 7.2 Summary

3.8 chapter 8: Metaclasses – Making Classes (Not Instances) Smarter

Dynamically creating classes 1026 Abstract classes using collections.abc 1030 Automatically registering a plugin system 1037 Order of operations when instantiating classes 1043 Storing class attributes in definition order 1048 Summary

3.8.1 8.1 Dynamically creating classes

8.1.1 Linux

Automatic Install Script

```
$ sudo wget -qO- https://get.docker.com/ | sh
```

remove hell-world

```
$ sudo docker rm `sudo docker ps -aq`
$ sudo docker rmi hello-world
```

.

3.8.2 8.2 Abstract classes using collections.abc

3.8.3 8.3 Automatically registering a plugin system

3.8.4 8.4 Order of operations when instantiating classes

3.8.5 8.5 Storing class attributes in definition order

3.8.6 8.6 Summary

3.9 chapter 9: Documentation – How to Use Sphinx and reStructured-Text

The reStructuredText syntax 1054 The Sphinx documentation generator 1069 Documenting code 1085 Summary

3.9.1 9.1 The reStructuredText syntax

9.1.1 Linux

Automatic Install Script

```
$ sudo wget -qO- https://get.docker.com/ | sh
```

remove hell-world

```
$ sudo docker rm `sudo docker ps -aq`
$ sudo docker rmi hello-world
```

.

3.9.2 9.2 The Sphinx documentation generator

3.9.3 9.3 Documenting code

3.9.4 9.4 Summary

3.10 chapter 10: Testing and Logging – Preparing for Bugs

Using examples as tests with doctest 1094 Testing with py.test 1110 Mock objects 1138 Logging 1141 Summary

3.10.1 10.1 Using examples as tests with doctest

10.1.1 Linux

Automatic Install Script

```
$ sudo wget -qO- https://get.docker.com/ | sh
```

remove hell-world

```
$ sudo docker rm `sudo docker ps -aq`
$ sudo docker rmi hello-world
```

.

3.10.2 10.2 Testing with py.test

3.10.3 10.3 Mock objects

3.10.4 10.4 Logging

3.10.5 10.5 Summary

3.11 chapter 11: Debugging – Solving the Bugs

Non-interactive debugging 1156 Interactive debugging 1168 Summary

3.11.1 11.1 Non-interactive debugging

11.1.1 Linux

Automatic Install Script

```
$ sudo wget -qO- https://get.docker.com/ | sh
```

remove hell-world

```
$ sudo docker rm `sudo docker ps -aq`
$ sudo docker rmi hello-world
```

.

3.11.2 11.2 Interactive debugging

3.11.3 11.3 Summary

3.12 chapter 12: Performance – Tracking and Reducing Your Memory and CPU Usage

What is performance? 1182 Timeit – comparing code snippet performance 1183 cProfile – finding the slowest components 1187 Line profiler 1197 Improving performance 1199 Memory usage 1205 Performance monitoring 1218 Summary

3.12.1 12.1 What is performance?

12.1.1 Linux

Automatic Install Script

```
$ sudo wget -qO- https://get.docker.com/ | sh
```

remove hell-world

```
$ sudo docker rm `sudo docker ps -aq`
$ sudo docker rmi hello-world
```

.

3.12.2 12.2 Timeit – comparing code snippet performance

3.12.3 12.3 cProfile – finding the slowest components

3.12.4 12.4 Line profiler

3.12.5 12.5 Improving performance

3.12.6 12.6 Memory usage

3.12.7 12.7 Performance monitoring

3.12.8 12.8 Summary

3.13 chapter 13: Multiprocessing – When a Single CPU Core Is Not Enough

Multithreading versus multiprocessing 1221 Hyper-threading versus physical CPU cores 1224 Creating a pool of workers 1226 Sharing data between processes 1228 Remote processes 1229 Summary

3.13.1 13.1 Multithreading versus multiprocessing

13.1.1 Linux

Automatic Install Script

```
$ sudo wget -qO- https://get.docker.com/ | sh
```

remove hell-world

```
$ sudo docker rm `sudo docker ps -aq`
$ sudo docker rmi hello-world
```

.

3.13.2 13.2 Hyper-threading versus physical CPU cores

3.13.3 13.3 Creating a pool of workers

3.13.4 13.4 Sharing data between processes

3.13.5 13.5 Remote processes

3.13.6 13.6 Summary

3.14 chapter 14: Extensions in C/C++, System Calls, and C/C++ Libraries

Introduction 1239 Calling C/C++ with ctypes 1242 CFFI 1249 Native C/C++ extensions 1252 Summary

3.14.1 14.1 Introduction

14.1.1 Linux

Automatic Install Script

```
$ sudo wget -qO- https://get.docker.com/ | sh
```

remove hell-world

```
$ sudo docker rm `sudo docker ps -aq`
$ sudo docker rmi hello-world
```

.

3.14.2 14.2 Calling C/C++ with ctypes

3.14.3 14.3 CFFI

3.14.4 14.4 Native C/C++ extensions

3.14.5 14.5 Summary

3.15 chapter 15: Packaging – Creating Your Own Libraries or Applications

Installing packages 1265 Setup parameters 1266 Packages 1270 Entry points 1270 Package data 1274 Testing packages 1275 C/C++ extensions 1279 Wheels – the new eggs 1282 Summary

3.15.1 15.1 Installing packages

15.1.1 Linux

Automatic Install Script

```
$ sudo wget -qO- https://get.docker.com/ | sh
```

remove hell-world

```
$ sudo docker rm `sudo docker ps -aq`
$ sudo docker rmi hello-world
```

.

3.15.2 15.2 Setup parameters

3.15.3 15.3 Packages

3.15.4 15.4 Entry points

3.15.5 15.5 Package data

3.15.6 15.6 Testing packages

3.15.7 15.7 C/C++ extensions

3.15.8 15.8 Wheels – the new eggs

3.15.9 15.9 Summary