Python Snippets

A Collection of Useful Python Snippets with Explanations

Florian Dahlitz

June 28, 2019

Contents

Chapter 1

Standard Library

This chapter contains the Python snippets from the *standard_lib* directory. Additionally, explanations are added and possible usages are described.

1.1 Abstract Base Class (ABC)

The `abc` module provides the infrastructure for defining abstract base classes (ABCs) in Python. In this section you will find a collection of snippets being useful when dealing with ABCs.

1.1.1 General Usage

Listing 1.1 shows the general usage of ABCs by example.

```
from abc import ABCMeta, abstractmethod


class BaseClass(metaclass=ABCMeta):
    @abstractmethod
    def foo(self):
        pass

    @abstractmethod
    def bar(self):
        pass


class ConcreteClass(BaseClass):
    def foo(self):
        pass

    def bar(self):
        pass


instance = ConcreteClass()
```

Listing 1.1: abc_class.py

The snippets contains two classes: `BaseClass` and `ConcreteClass`, whereas `BaseClass` is the ABC of

`ConcreteClass`. Both methods `foo` and `bar` are decorated with `@abstractmethod` and have to be implemented by the classes inheriting from `BaseClass`. Furthermore, as `BaseClass` is an ABC, it can't be instantiated.

1.2 AsyncIO

Since Python 3.5 the module `asyncio` provides the keywords `async` and `await`. This section provides some useful snippets when operating with them.

1.2.1 Time Saving

Using `async` and `await` will save you some time especially when requesting data from the internet. To keep things simple let's simulate requests against a certain API with `time.sleep()`. Listing 1.2 shows a sample program.

```
"""
Async snippet demonstrating the usage of
    ↪ time.sleep()
"""
import asyncio
import time
```

```
from datetime import datetime


async def custom_sleep():
    print("SLEEP", datetime.now())
    time.sleep(1)


async def factorial(name, number):
    f = 1
    for i in range(2, number + 1):
        print(f"Task {name}: Compute
            ↪ factorial({i})")
        await custom_sleep()
        f *= i
    print(f"Task {name}: factorial({
        ↪ number}) is {i}\n")


start = time.time()
loop = asyncio.get_event_loop()

tasks = [
    asyncio.ensure_future(factorial("A",
        ↪  3)),
    asyncio.ensure_future(factorial("B",
        ↪  4)),
]
```

```
loop.run_until_complete(asyncio.wait(
    ↪ tasks))
loop.close()

end = time.time()
print(f"Total time: {end - start}")
```

Listing 1.2: async_sleep_sync.py

Running this snippet takes around five seconds. Let's modify it a little bit to run the tasks asynchronously, which is shown in Listing 1.3.

```
"""
Async snippet demonstarting the usage of
    ↪ asyncio.sleep()
"""
import asyncio
import time
from datetime import datetime


async def custom_sleep():
    print(f"SLEEP {datetime.now()}\n")
    await asyncio.sleep(1)
```

```
async def factorial(name, number):
    f = 1
    for i in range(2, number + 1):
        print(f"Task {name}: Compute
            ↪ factorial({i})")
        await custom_sleep()
        f *= i
    print(f"Task {name}: factorial({
        ↪ number}) is {f}\n")


start = time.time()
loop = asyncio.get_event_loop()

tasks = [
    asyncio.ensure_future(factorial("A",
        ↪  3)),
    asyncio.ensure_future(factorial("B",
        ↪  4)),
]
loop.run_until_complete(asyncio.wait(
    ↪ tasks))
loop.close()

end = time.time()
print(f"Total time: {end - start}")
```

Listing 1.3: async_sleep_async.py

After the modifications it lasts around three seconds.

1.3 Playing with Bits

In this section you will find a collection of snippets corresponding to bit manipulation.

1.3.1 Bit Flipper

The `bit_flipper` function gives you a poor man's encryption. Making use of a salt, it flips certain bits of each charcter. The resulting string hides the information from the original one, at least for humans. Applying the function a second time returns the original string.

```
# coding: utf-8

# https://forum.omz-software.com/topic
    ↪ /2943/trying-to-make-an-encrypted-
    ↪ message-system
# a poor man's encryption


def bit_flipper(s, salt=1):
    return "".join([chr(ord(x) ^ salt)
        ↪ for x in s])
```

```
salt = 1  # try 1, 6, 7
# for instance, salt = 2 gives you an
    ↪ encrypted string with no printable
    ↪  chars
# (disappearing ink)!
s = "Pythonista rules!"
print(s)
s = bit_flipper(s, salt)
print(s)
s = bit_flipper(s, salt)
print(s)
```

Listing 1.4: bit_flipper.py

The output will look like the following:

```
$ python bit_flipper.py
Pythonista rules!
Qxuinohru‘!stmdr
Pythonista rules!
```

Listing 1.5: Output of the bit flipper

1.3.2 A Simple Bitmask

The `BitMask` class represents a simple bit mask. It has methods representing all the bitwise operations plus some additional features. The methods return a new BitMask object or a boolean result. See the bits module for more information about the operations provided.

```
#! /bin/env python3


class BitMask(int):
    def AND(self, bm):
        return BitMask(self & bm)

    def OR(self, bm):
        return BitMask(self | bm)

    def XOR(self, bm):
        return BitMask(self ^ bm)

    def NOT(self):
        return BitMask(~self)

    def shiftleft(self, num):
        return BitMask(self << num)
```

```
    def shiftright(self, num):
        return BitMask(self > num)

    def bit(self, num):
        mask = 1 << num
        return bool(self & mask)

    def setbit(self, num):
        mask = 1 << num
        return BitMask(self | mask)

    def zerobit(self, num):
        mask = ~(1 << num)
        return BitMask(self & mask)

    def listbits(self, start=0, end=-1):
        end = end if end < 0 else end +
            ↪ 2
        return [int(c) for c in bin(self
            ↪ )[start + 2 : end]]
```

Listing 1.6: bitmask.py

1.4 Search Algorithms

This section contains implementations of certain search algorithms.

1.4.1 Binary Search

The binary search algortihm is a search algorithm, that finds the position of a target value within a sorted array. It compares the target value to the middle element of the array. If they are not equal, the half in which the target cannot lie is eliminated and the search continues on the remaining half, again taking the middle element to compare to the target value, and repeating this until the target value is found.

```
import random


def binary_search(lst, integer):

    lower_bound = 0
    upper_bound = len(lst) - 1
    number_guesses = 0

    while True:
```

```
        index = (lower_bound +
           ↪ upper_bound) // 2

        guess = lst[index]
        number_guesses += 1

        if guess == integer:
            break

        if guess > integer:
            upper_bound = index - 1
        else:
            lower_bound = index + 1

    return number_guesses


print("Number Guesses")

for _ in range(30):
    integers = random.sample(range
       ↪ (50000), 10000)

    integer_to_find = random.choice(
       ↪ integers)

    sorted_integers = sorted(integers)
```

```
guesses = binary_search(
    ↪ sorted_integers,
    ↪ integer_to_find)

print("{:6} {:7}".format(
    ↪ integer_to_find, guesses))
```

Listing 1.7: binary_search.py

1.5 Dealing with builtins

Sometimes it can be helpful to manipulate builtins or to find a better way to deal with them. This section will show you ways to interact with builtins you may not thought of.

1.5.1 Change Behaviour

In some situations it can be helpful to extend the functionality of a certain builtin. Keep in mind, that this can be dangerous if it's not documented and is done in a global scope!

```
_print = print
```

```
def print(*args, **kwargs):
    nargs = (*args, "\nNumber of
       ↪ arguments:", len(args))
    _print(*nargs, **kwargs)


print("One", "Two", "Three")
```

Listing 1.8: builtins_manipulation.py

The Listing first assigns the builtin **print** function to a variable called **_print**. After that, a custom **print** function is defined shadowing the builtin one. The functionality is extended by displaying the number of arguments before printing the actual output. You can find the output in the following Listing.

```
$ python builtins_manipulation.py
One Two Three
Number of arguments: 3
```

Listing 1.9: Output of builtins_manipulation.py

1.5.2 Capture Output

It may be helpful to capture and redirect the output of certain functions. For instance you don't want to send the output of the builtin **help** function to stdout but want to redirect it to a file. The following Listing shows you three ways how to capture and redirect the output of functions.

```
import contextlib
import io
import sys


# Option 1
output_stream = io.StringIO()
with contextlib.redirect_stdout(
    ↪ output_stream):
    help(pow)

output_stream = output_stream.getvalue()
print("value:", output_stream)


# Option 2
with open("help.txt", "w") as f:
    with contextlib.redirect_stdout(f):
```

```
        help(pow)


# Option 3
with contextlib.redirect_stdout(sys.
    ↪ stderr):
    help(pow)
```

Listing 1.10: capture_output.py

The first option saves the output to a `StringIO` object. The value can be accessed using `.getvalue()`.

The second option can be used to save the output to a specified file. In this case we save the output to `help.txt`.

Last but not least we are redirecting the output to stderr.

1.6 String Operations

In the following section you will find useful snippets when dealing with strings.

1.6.1 Check for String Pattern

Assuming you have a list of strings and you want to check them for multiple pattern. The following snippet solves

this issue by using a fairly simple list comprehension.

```
"""
Check for multiple string patterns.
"""
lst = ["hello", "fafaea", "hello world",
    ↪  "xxx world", "zzz"]
patterns = ["hello", "world"]

result = [item for item in lst if any(
    ↪ pattern in item for pattern in
    ↪ patterns)]

print(result)
```

Listing 1.11: check_pattern.py

The output after running the snippet is something like that:

```
$ python check_pattern.py
['hello', 'hello world', 'xxx world']
```

Listing 1.12: Output of check_pattern.py

1.6.2 Compare Strings

If you want to compare two strings and want to know „how equal they are“, you can make use of the `SequenceMatcher`.

```
from difflib import SequenceMatcher


s1 = "This is my string"
s2 = "This is my new string"

s = SequenceMatcher(None, s1, s2)

print(round(s.ratio(), 3))

for block in s.get_matching_blocks():
    a, b, size = block
    print(f"a[{a}] and b[{b}] match for
       ↪ {size} elements")
```

Listing 1.13: compare_strings.py

It will split the strings into matching blocks and return how many characters of the blocks are matching. Furthermore, you get a float value representing the overall matching. The output is shown in the following List-

ing.

```
$ python compare_strings.py
0.895
a[0] and b[0] match for 11 elements
a[11] and b[15] match for 6 elements
a[17] and b[21] match for 0 elements
```

Listing 1.14: Output of compare_strings.py

1.6.3 Fill Strings

You can use the `zfill` string method to fill a string with zeros if the provided maximum length isn't already reached.

```
for i in [1, 11, 222, "aaa", 1234]:
    print(str(i).zfill(4))
```

Listing 1.15: fill_zeros.py

The outpur is shown in the following Listing.

```
$ python fill_zeros.py
```

```
0001
0011
0222
0aaa
1234
```

Listing 1.16: Output of fill_zeros.py

1.6.4 Parse Query String

You can use the builtin **urllib** module to parse query strings.

```
from urllib.parse import parse_qs


input_string = "host=171.25.2.7&port
   ↪ =8080&port=8081"
result = parse_qs(input_string)
print(result)
```

Listing 1.17: parse_query_string.py

The provided snippets will output the following:

```
$ python parse_query_string.py
{'host': ['171.25.2.7'], 'port': ['8080'
    ↪ , '8081']}
```

Listing 1.18: Output of parse_query_string.py

1.6.5 Print Numbers Human-Friendly

You can print numbers in a human-friendly way using the builtin **format** function.

```
n = 123123123
print(format(n, ","))
```

Listing 1.19: print_human_friendly_numbers.py

```
$ python print_human_friendly_numbers.py
123,123,123
```

Listing 1.20: Output of print_human_friendly_numbers.py

1.6.6 Split Strings Preserving Substrings

In some cases you're splitting a string into the words it contains, but you want to keep substrings. Fortunately, Python has a module called `shlex` providing a `split` function keeping substrings as one.

```
import shlex


sample = 'This is "Berlin Cafe"'
split_str = sample.split()
shlex_str = shlex.split(sample)
some = shlex.quote(sample)

print(f"Split string output: {split_str}
   ↪ ")
print(f"Shlex string outpur: {shlex_str}
   ↪ ")
```

Listing 1.21: split_preserving_sub-strings.py

```
$ python split_preserving_sub-strings.py
Split string output: ['This', 'is', '"
```

```
   ↪ Berlin', 'Cafe"']
Shlex string outpur: ['This', 'is', '
   ↪ Berlin Cafe']
```

Listing 1.22: Output of split_preserving_sub-strings.py

1.6.7 capitalize() vs. title()

The following Listings shows the difference between `capitalize` ↪ and `title`

```
string = "i love coffee"

print(f"capitalize(): {string.capitalize
   ↪ ()}")
print(f"title(): {string.title()}")
```

Listing 1.23: string_capitalize.py

```
$ python string_capitalize.py
capitalize(): I love coffee
title(): I Love Coffee
```

Listing 1.24: Output of string_capitalize.py

1.7 Dictionaries

When dealing with dictionaries the following snippets might be helpful - or just fascinating.

1.7.1 Crazy Dictionary Expression

The following snippet shows you a crazy dictionary expression. Maybe the most craziest dictionary expression you've seen so far.

```
from pprint import pprint

crazy_dict = {True: "yes", 1: "no", 1.0:
    ↪  "maybe"}
pprint(crazy_dict)
```

Listing 1.25: crazy_dict_expression.py

If you're not familiar with dictionary keys, the following output might confuse you.

```
$ python crazy_dict_expression.py
{True: 'maybe'}
```

Listing 1.26: Output of crazy_dict_expression.py

Dictionary keys are compared by their hash values. As `True`, `1` and `1.0` have the same hash values, the keys value gets overwritten. `True` is inserted as no key with the same hash value exists in the dictionary so far. Inserting `1` will lead to an overwriting of the value of `True` as both share the same hash value. This results in the following dictionary:

```
{True: "no"}
```

Last but not least `1.0` with its corresponding value is inserted. The behaviour is similar to the insertion of `1` resulting in the final output shown in Listing 1.26.

1.7.2 Emulate switch-case

A `switch-case` statement doesn't exist in Python, but you can easily emulate it using a dictionary. By using `.get()` you can provide a default return value if a `KeyError` is raised.

```
import operator


dispatch_dict = {
```

```
    "add": operator.add,
    "sub": operator.sub,
    "mul": operator.mul,
    "div": operator.floordiv,
}


def dispatch_func(op, x, y):
    return dispatch_dict.get(op, lambda:
        ↪  None)(x, y)


print(f"dispatch_func('mul', 4, 5) = {
   ↪ dispatch_func('mul', 4, 5)}")
```

Listing 1.27: emulate_switch_case.py

1.7.3 Merge Abitrary Number of Dicts

You can use the following snippet to merge an arbitrary number of dictionaries using the ** operator.

```
dict1 = {"a": 1, "b": 2}
dict2 = {"b": 3, "c": 4, 3: 9}
dict3 = {"a": 2}
dict4 = {"d": 8, "e": 1}
```

```
print("Result of merging dict 1-4:", {**
    ↪ dict1, **dict2, **dict3, **dict4})
```

Listing 1.28: merge_arbitrary_number_of_dicts.py

Note that later dictionaries are overwriting existing key-value-pairs.

```
$ python merge_arbitrary_number_of_dicts
    ↪ .py
Result of merging dict 1-4: {'a': 2, 'b'
    ↪ : 3, 'c': 4, 3: 9, 'd': 8, 'e': 1}
```

Listing 1.29: Output of merge_arbitrary_number_of_dicts.py

1.7.4 MultiDict with Default Values

Sometimes you simply want to store several values for a single key. `MultiDict` is the perfect data structure for that. However, a `MultiDict` implementation is not provided by the Python standard library. You can implement your own by using lists as dict values. However, when inserting a value to a key, which doesn't exist so

far, a `KeyError` is raised. Using a `defaultdict` solves the problem as a new empty list is added if the key not already exists.

```
from collections import defaultdict


rd = defaultdict(list)

for name, num in [("ichi", 1), ("one",
    ↪ 1), ("uno", 1), ("un", 1)]:
    rd[num].append(name)

print(rd)
```

Listing 1.30: multidict_with_default_init.py

The output shows what we expect.

```
$ python multidict_with_default_init.py
defaultdict(<class 'list'>, {1: ['ichi',
    ↪  'one', 'uno', 'un']})
```

Listing 1.31: Output of multidict_with_default_init.py

1.7.5 Overwrite Dictionary Values

Loving Python 3.5 PEP 448 for overwriting a dictionary of default values, you can use the following snippet to do exactly that! It makes use of the very same operator used when merging dictionaries: **.

```
defaults = {"lenny": "white", "carl": "
   ↪ black"}
overwrite = {"lenny": "yellow"}
new = {**defaults, **overwrite}

print(new)
```

Listing 1.32: overwrite_dictionary.py

You may use it when you provide a dictionary containing default configuration settings and want to overwrite them with custom settings.

```
$ python overwrite_dictionary.py
{'lenny': 'yellow', 'carl': 'black'}
```

Listing 1.33: Output of overwrite_dictionary.py

1.7.6 Pass Multiple Dicts as Argument

If you want to pass multiple dicts as argument for a certain function, you can make use of the ** operator for auto-unpacking. However, there are two ways to achieve that depending on whether you want dictionary keys to be overwritten or not.

```
from collections import ChainMap


def f(a, b, c, d):
    print(a, b, c, d)


d1 = dict(a=5, b=6)
d2 = dict(b=7, c=8, d=9)

# The old may without overwriting
    ↪ previous values
f(**ChainMap(d1, d2))

# Since Python 3.5 allowing value
    ↪ overwriting
f(**{**d1, **d2})
```

Listing 1.34: pass_multiple_dicts.py

Using a `ChainMap` doesn't overwrite earlier specified key-value-pairs. The second option overwrites key-value-pairs.

```
$ python pass_multiple_dicts.py
5 6 8 9
5 7 8 9
```

Listing 1.35: Output of pass_multiple_dicts.py

1.7.7 Default Configuration

The following snippetshows you, how you can provide a default config, overwrite it with a config file and overwrite the result with command-line arguments. All this can be achieved by using `ChainMap`.

```
from collections import ChainMap


def main():
    command_line_args =
       ↪ get_command_line_args()
    config_file = get_config()
```

```
    default_config = get_default_config
       ↪ ()

    config_dict = ChainMap(
       ↪ command_line_args, config_file
       ↪ , default_config)


def get_command_line_args() -> dict:
    pass


def get_config() -> dict:
    pass


def get_default_config() -> dict:
    pass


if __name__ == "__main__":
    main()
```

Listing 1.36: provide_default_config_values.py

1.7.8 Keep Original After Updates

You may want to update a dictionaries values without loosing the original values. This can be achieved by using a `ChainMap`. Therefore, we chain an empty dictionary and the one containing the original values. Now we are only operating with the `ChainMap`.

```
from collections import ChainMap


population = {"italy": 60, "japan": 127,
    ↪  "uk": 65}

changes = dict()
editable = ChainMap(changes, population)

print(f"Before update: {editable['japan
    ↪ ']}")

editable["japan"] += 1
print(f"Updated population: {editable['
    ↪ japan']}")
print(f"Original population: {population
    ↪ ['japan']}")

print(f"Changes: {changes.keys()}")
```

```
print(f"Did not changed: {population.
    ↪ keys() - changes.keys()}")
```

Listing 1.37: save_dict_update_without_loosing_original.py

If we insert new values, they are added to the `changes` dict. If we try to retrieve values, which aren't part of the `changes` dict, we fall back and use the once from the original dict.

```
$ python
    ↪ save_dict_update_without_loosing_original
    ↪ .py
Before update: 127
Updated population: 128
Original population: 127
Changes: dict_keys(['japan'])
Did not changed: {'italy', 'uk'}
```

Listing 1.38: Output of save_dict_update_without_loosing_original.py

1.7.9 Update Dict Using Tuples

You can update a dictionaries values by using tuples.

```
a = {1: "1"}
b = [(2, "2"), (3, "3")]

a.update(b)

print(a)
```

Listing 1.39: update_dict_using_tuples.py

```
$ python update_dict_using_tuples.py
{1: '1', 2: '2', 3: '3'}
```

Listing 1.40: Output of update_dict_using_tuples.py

1.7.10 Create A Dict Based On Lists

Let's assume you have two lists and you want to create a dictionary out of them. The first list might contain the letters 1-26 and the second list might contain the characters a-z. So you want to map a character to each numbers to finally end up with a dictionary representing the alphabet. This recipe shows you how to achieve that in a very simple manner.

```
import json


nums = [1, 2, 3]
chars = ["a", "b", "c"]

alph = dict(zip(nums, chars))
print(json.dumps(alph, indent=4,
   ↪ sort_keys=True))
```

Listing 1.41: dict_based_on_lists.py

As we are using `json.dumps` with the argument `sortkeys ↪ =True`, the final dictionary is sorted and looks like this:

```
$ python dict_based_on_lists.py
{
    "1": "a",
    "2": "b",
    "3": "c"
}
```

Listing 1.42: Output of dict_based_on_lists.py

1.8 Decorators

A collection of quite useful decorators you can use in your projects.

1.8.1 Deprecation Decorator

Let's assume you want to remove a certain function or method. Instead of just removing it, you want to signal your users that the function or method is deprecated, so they know, it will be removed in a future release. You can write your custom deprecation decorator printing a `DeprecationWarning` whenever the deprecated function or method is invoked. Listing 1.43 shows you a sample implementation of the decorator as well as possible usages.

```
import warnings


def deprecated(func):
    """This is a decorator which can be
       ↪ used to wmark functions as
       ↪ deprecated.
    It will result in a warning being
       ↪ emitted when the function is
       ↪ used."""
```

```
    def new_func(*args, **kwargs):
        warnings.warn(
            f"Call to deprecated
               ↪ function {func.
               ↪ __name__}.", category=
               ↪ DeprecationWarning
        )

        return func(*args, **kwargs)

    new_func.__name__ = func.__name__
    new_func.__doc__ = func.__doc__
    new_func.__dict__.update(func.
       ↪ __dict__)

    return new_func


# === Example usages ===


@deprecated
def some_old_function(x, y):
    return x + y


class SomeClass:
```

```
    @deprecated
    def some_old_method(self, x, y):
        return x + y


some_old_function(1, 2)
example = SomeClass()
example.some_old_method(1, 2)
```

Listing 1.43: deprecated_decorator.py

Everytime the deprecated function or method is called, a `DeprecationWarning` is printed.

```
$ python deprecated_decorator.py
<your_path>/deprecated_decorator.py:10:
   ↪ DeprecationWarning: Call to
   ↪ deprecated function
   ↪ some_old_function.
f"Call to deprecated function {func.
   ↪ __name__}.", category=
   ↪ DeprecationWarning
<your_path>/deprecated_decorator.py:10:
   ↪ DeprecationWarning: Call to
   ↪ deprecated function
   ↪ some_old_method.
f"Call to deprecated function {func.
```

```
    ↪ __name__}.", category=
    ↪ DeprecationWarning
```

Listing 1.44: Output of deprecated_decorator.py

1.8.2 Keep Function Metadata

The original functions metadata are lost when using a decorator. You can keep them using the `@wraps` decorator provided by the `functools` module.

```
from functools import wraps


def tags(tag_name):
    """Wraps another function"""

    def tags_decorator(func):
        @wraps(func)
        def func_wrapper(name):
            return f"<{tag_name}>{func(
                ↪ name)}</{tag_name}>"

        return func_wrapper

    return tags_decorator
```

```
@tags("p")
def get_text(name):
    """Returns some text"""
    return "Hello " + name


print(get_text("World"))

print(get_text.__name__)
print(get_text.__doc__)
```

Listing 1.45: keep_metadata_on_decorator_usage.py

The output shows the result we would expect from a decorated function.

```
$ python
   ↪ keep_metadata_on_decorator_usage.
   ↪ py
<p>Hello World</p>
get_text
Returns some text
```

Listing 1.46: Output of keep_metadata_on_decorator_usage.py

1.8.3 Trace Decorator

This snippet contains a custom trace decorator revealing a functions flow.

```
import functools


def trace(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        print("-" * 20)
        print(f"TRACE: calling {func.
            ↪ __name__}() " f"with {args
            ↪ },ă{kwargs}")

        original_result = func(*args, **
            ↪ kwargs)

        print(f"TRACE: {func.__name__}()
            ↪  " f"returned {
            ↪ original_result!r}")
        print("-" * 20)

        return original_result

    return wrapper
```

```
@trace
def greet(name, phrase):
    return f"{name}, {phrase}"


print(greet("Florian", "Nice to see you!
    ↪ "))
```

Listing 1.47: trace_decorator.py

The output is shown in the following Listing.

```
$ python trace_decorator.py
```

```
TRACE: calling greet() with ('Florian',
    ↪ 'Nice to see you!'),ă{}
TRACE: greet() returned 'Florian, Nice
    ↪ to see you!'
```

```
Florian, Nice to see you!
```

Listing 1.48: Output of trace_decorator.py

1.9 Bytecode

This section contains snippets for investigating Pythons bytecode.

1.9.1 Disassemble Bytecode - String Conversion

Ever wanted to know what a string conversion using f-strings or **str()** looks like in bytecode? The following snippet shows you exactly that!

```
import dis


def f_string(number: int):
    return f"{number}"


def str_string(number: int):
    return str(number)


print("============================== f-
   ↪ strings
   ↪ ==============================")
dis.dis(f_string)
```

```
print("================================
    ↪ str()
    ↪ ================================")
dis.dis(str_string)
```

Listing 1.49: disassemble_bytecode.py

For displaying the bytecode of a function the builtin `dis` module is used.

```
$ python disassemble_bytecode.py
============================== f-strings
    ↪  ==============================
  5           0 LOAD_FAST
      ↪                 0 (number)
              2 FORMAT_VALUE
                ↪              0
              4 RETURN_VALUE
================================ str()
    ↪ ================================
  9           0 LOAD_GLOBAL
      ↪              0 (str)
              2 LOAD_FAST
                ↪                  0 (
                ↪ number)
              4 CALL_FUNCTION
                ↪            1
```

```
6 RETURN_VALUE
```

Listing 1.50: Output of disassemble_bytecode.py

1.9.2 Human Readable Bytecode

If you are using Pythons `dis` module to get some insights into what's happening behind the scenes, you may like this recipe as it provides a way to print the output in a much more human readable way.

```
from struct import unpack
from dis import opmap

reverse_opmap = {v: k for k, v in opmap.
    ↪ items()}


def foo():
    pass


foo_code = foo.__code__.co_code
for pos in range(0, len(foo_code), 2):
    inst = unpack("BB", foo_code[pos :
        ↪ pos + 2])
```

```
    print(f"{reverse_opmap[inst[0]]}: {
        ↪ inst[1]}")
```

Listing 1.51: human_readable_bytecode.py

```
$ python human_readable_bytecode.py
LOAD_CONST: 0
RETURN_VALUE: 0
```

Listing 1.52: Output of human_readable_bytecode.py

Note: Even though we didn't put a return statement at the end of `foo`, Python added it. That's because every function in Python needs a return statement (specified by the underlying protocols).

1.10 Lists

Provide useful tips for dealing with lists.

1.10.1 Flatten a List

Sometimes you have a nested list and just want to flatten it. Here's a small snippet revealing how to achieve right that.

```
from collections.abc import Iterable


def flatten(input_arr, output_arr=None):
    if output_arr is None:
        output_arr = []
    for ele in input_arr:
        if isinstance(ele, Iterable):
            flatten(ele, output_arr)  #
                ↪ tail-recursion
        else:
            output_arr.append(ele)  #
                ↪ produce the result

    return output_arr


sample_list = [1, [2], [[3, 4], 5], 6]
print(flatten(sample_list))
```

Listing 1.53: flatten.py

```
$ python flatten.py
[1, 2, 3, 4, 5, 6]
```

Listing 1.54: Output of flatten.py

1.10.2 Priority Queue

Image you have a sports tournament (e.g. table tennis or basketball). Now you want to get the player or team with the most points. You could store all the information in a list, but if new items are added, you need to resort it. This can take a significant time amount if the data set keeps growing. You can make use of a `Heap` data structure to implement your own priority queue to autosort the data for you.

```
import time
import heapq


class PriorityQueue:
    def __init__(self):
        self._q = []

    def add(self, value, priority=0):
        heapq.heappush(self._q, (
            ↪ priority, time.time(),
```

```
            ↪ value))

    def pop(self):
        return heapq.heappop(self._q)
            ↪ [-1]


f1 = lambda: print("hello")
f2 = lambda: print("world")

pq = PriorityQueue()
pq.add(f2, priority=1)
pq.add(f1, priority=0)

print(pq.pop()())
print(pq.pop()())
```

Listing 1.55: priority_queue.py

As the snippet provides a sample usage, the following Listing shows you the output.

```
$ python priority_queue.py
hello
None
world
None
```

Listing 1.56: Output of priority_queue.py

1.10.3 Remove Duplicates

Remove duplicates from a list but keeping the order by using `OrderedDict`.

```
from collections import OrderedDict


a = ["foo", "Alice", "bar", "foo", "Bob"
    ↪ ]
print(f"List with duplicates: {a}")

a = list(OrderedDict.fromkeys(a).keys())
print(f"Without duplicates: {a}")
```

Listing 1.57: remove_duplicates_list.py

```
$ python remove_duplicates_list.py
List with duplicates: ['foo', 'Alice', '
    ↪ bar', 'foo', 'Bob']
```

```
Without duplicates: ['foo', 'Alice', '
    ↪ bar', 'Bob']
```

Listing 1.58: Output of remove_duplicates_list.py

1.10.4 Unpacking Lists Using * Operator

Assuming you have a list with more elements than you have variables to store the values in. You can use the * operator when unpacking a list to store a partial list in one variable.

```
a, *b, c = [1, 2, 3, 4, 5]

print(f"a = {a}")
print(f"b = {b}")
print(f"c = {c}")
```

Listing 1.59: list_unpacking.py

```
$ python list_unpacking.py
a = 1
b = [2, 3, 4]
```

```
c = 5
```

Listing 1.60: Output of list_unpacking.py

1.10.5 Remove Elements Not Matching Pattern

Let's assume you have a list of certain data and want to remove all elements in it, which are not matching a certain pattern. You could make use of the built-in `list.remove()` method, which removes the first match of and then shifts all subsequent data one position to the left. So, if you want to remove all, you need to loop over it. However, this approach gives quadratic behaviour. The following Listing shows you a much better way to achieve this:

```
data = [1, 2, 3, 4, 1, 2, 3, 4]
target = 1

print(f"Original: {data}")

data[:] = [elem for elem in data if elem
    ↪  != target]

print(f"New: {data}")
```

Listing 1.61: remove_elements_list.py

This approach is not only faster, but creates a new, distinct list and then replaces the old contents all at once.

1.11 Files

In this section you will find a collection of snippets revealing tips for interacting with files.

1.11.1 Hash a File

To ensure a files integrity, you can hash the file and compare it with other hashes. The following Listing shows you how to hash a certain file using `MD5` and `SHA1`. Both hashes are printed to stdout as well as the name of the hashed file.

```
"""Reference: https://medium.com/
    ↪ ediblesec/building-a-hashing-tool-
    ↪ with-python-3afe34db74e5"""
import os
import hashlib
```

```
import argparse

parser = argparse.ArgumentParser()
parser.add_argument("file")
args = parser.parse_args()

md5 = hashlib.md5()
sha1 = hashlib.sha1()

try:
    with open(args.file, "rb") as f:
        buf = f.read()
        md5.update(buf)
        sha1.update(buf)

    print(f"Filename: {os.path.basename(
        ↪ args.file)}")
    print(f"MD5-Hash: {md5.hexdigest()}"
        ↪ )
    print(f"SHA1-Hash: {sha1.hexdigest()
        ↪ }")
except FileNotFoundError as e:
    print(e)
```

Listing 1.62: hash_file.py

1.11.2 Read Files using Iterator

Reveals the usage of iterators to read in a file. It's useful when dealing with large files. If not using iterators, the whole file is loaded into memory at once (think of several gigabyte huge files). If using iterators, only the next line is loaded.

```
print(next(open("huge_log_file.txt")))
```

Listing 1.63: read_files_using_iterator.py

1.11.3 File Matching Using `fnmatch`

Matching certain strings is easy if you use Pythons built-in `fnmatch` module. It even provides you functionality to translate the easy to use `fnmatch` patterns into regular expressions. The following recipe shows you a sample usage filtering markdown files and `git`-related files in your current working directory.

```
import fnmatch
import os
import re
```

```
reg = "({})|({})".format(fnmatch.
    ↪ translate("*.md"), fnmatch.
    ↪ translate("*.git*"))
markdown_files = [f for f in os.listdir
    ↪ () if re.match(reg, f)]

print(markdown_files)
```

Listing 1.64: file_matching_regex.py

1.12 Context Manager

In this section you will find a collection of self-implemented context manager.

1.12.1 Open Multiple Files

Ever wanted to save a certain text to multiple files? Well, with this context manager you can open multiple files at once and write a specified text to them.

```
from contextlib import ExitStack,
    ↪ contextmanager
```

```
@contextmanager
def multi_open(paths, mode="r"):
    with ExitStack() as stack:
        yield [stack.enter_context(open(
            ↪ path, mode)) for path in
            ↪ paths]


paths = (f"file_{n}.txt" for n in range
    ↪ (10))

with multi_open(paths, "w") as files:
    for file in files:
        file.write("Enter some text to
            ↪ several files.")
```

Listing 1.65: multi_open_files.py

If you run the snippet, it will create ten text files all containing the same text.

1.12.2 Temporal SQLite Table

If you are working with `sqlite`, you may find this context manager helpful. It creates a temporal table you can interact with.

```
from sqlite3 import connect
from contextlib import contextmanager


@contextmanager
def temptable(cur):
    cur.execute("create table points(x
       ↪ int, y int)")
    try:
        yield
    finally:
        cur.execute("drop table points")


with connect("test.db") as conn:
    cur = conn.cursor()
    with temptable(cur):
        cur.execute("insert into points
           ↪ (x, y) values(1, 1)")
        cur.execute("insert into points
           ↪ (x, y) values(1, 2)")
        cur.execute("insert into points
           ↪ (x, y) values(2, 1)")
        for row in cur.execute("select x
           ↪ , y from points"):
            print(row)
        for row in cur.execute("select
```

```
        ↪ sum(x * y) from points"):
          print(row)
```

Listing 1.66: temptable_contextmanager.py

When leaving the `with`-statement, the table is deleted. That said, you can run the snippet as often as you like, the output remains the same.

```
$ python temptable_contextmanager.py
(1, 1)
(1, 2)
(2, 1)
(5,)
```

Listing 1.67: Output of temptable_contextmanager.py

1.12.3 Timing Context Manager

Sometimes you simply want to find out, how long a certain code block needs to be executed. To do that you have different options: Use a third-party package like `boxx` or create your own timing context manager based on the standard library. The following Listing shows you, how you can create your own timing context manager. In

specific, this approach makes use of a generator function (you can implement this context manager based on a class as well).

```
import time

from contextlib import contextmanager


@contextmanager
def timing(description: str):
    start = time.time()
    yield
    print(f"{description}: {time.time()
        ↪ - start}")


with timing("Time List Comprehension"):
    s = [x for x in range(10_000_000)]
```

Listing 1.68: timing_context_manager.py

As you can see, the code is pretty straightforward. An example output is shown below.

```
$ python timing_context_manager.py
```

```
Time List Comprehension:
   ↪ 0.5184009075164795
```

Listing 1.69: Output of timing_context_manager.py

1.13 Non-Categorized

This section includes all those snippets not belonging to any of the underlying categories.

1.13.1 Turtle

The `turtle` module provides turtle graphics primitives, in both object-oriented and procedure-oriented ways. Because it uses `tkinter` for the underlying graphics, it needs a version of Python installed with Tk support. The following Listing shows you a sample implementation of the `turtle` module drawing a dragon.

```
import turtle


turtle.tracer(False)

turtle.penup()
```

```
turtle.goto(-50, -50)
turtle.pendown()

for i in range(1, 2 ** 16):
    turtle.forward(3)
    if ((i & -i) << 1) & i:
        turtle.lt(90)
    else:
        turtle.rt(90)

turtle.done()
```

Listing 1.70: drawing_turtle.py

1.13.2 Function Parameters

There's not much to say about the following snippet. It shows the different usages of positional and keyword arguments. The last call shows you how *not* to call a function with positional and keyword arguments. So don't be confused if a `TypeError` is raised.

```
def main():
    positional("Cheese", 21, "Meat")
    optional_keyword("Cheese", 21, "Meat
        ↪ ")
```

```
    optional_keyword("Cheese", 21, arg3=
       ↪ "Meat")
    force_keyword("Cheese", 21, arg3="
       ↪ Meat")
    force_keyword("Cheese", 21, "Meat")


def positional(arg1: str, arg2: int,
   ↪ arg3: str):
    print(arg1, arg2, arg3)


def optional_keyword(arg1: str, arg2:
   ↪ int, arg3: str = None):
    print(arg1, arg2, arg3)


def force_keyword(arg1: str, arg2: int,
   ↪ *, arg3: str = None):
    print(arg1, arg2, arg3)


if __name__ == "__main__":
    main()
```

Listing 1.71: function_arguments.py

1.13.3 Password Input

Python has a module called `getpass`, which lets you take user input without printing the typed characters.

```
import getpass


pwd = getpass.getpass()
print(pwd)
```

Listing 1.72: get_password_input.py

1.13.4 Hex Decode

You can decode hex-code in Python as follows.

```
print(bytes.fromhex("4d 65 72 72 79 20
    ↪ 43 68 72 69 73 74 6d 61 73 21"))
```

Listing 1.73: hex_decode.py

```
$ python hex_decode.py
b'Merry Christmas!'
```

Listing 1.74: Output of hex_decode.py

1.13.5 MicroWebServer

The implementation of a micro web server is stored in the `MicroWebServer.py` file. As this file is much longer than usual snippets and it's not meant to discussed it here, I'm referring the source code repo to see and test the program on your own.

1.13.6 Open Browser Tab

You can control a browser through the `webbrowser` module. If you want to open a new browser tab, you can simply run the code of the following Listing.

Note: Only browsers, which are part of the `PATH` variable, can be found

```
import webbrowser


controller = webbrowser.get("firefox")
controller.open_new_tab("https://
   ↪ stackoverflow.com")
```

Listing 1.75: open_browser_tab.py

1.13.7 Port Scanner

In the `port_scanner.py` file you find an implementation of a very basic port scanner. As the file is to large, it's not displayed here. Feel free to use the port scanner.

1.13.8 Reduce Memory Consumption - Customizing ___slots___

Every class has a `__slots__` attribute. This attribute is quite big by default. By specifying a custom `__slots__` attribute, you can reduce the memory consumption. As the snippet is quite large, I just refer to file in the repo: `reduce_memory_consumption.py`. However, I want to show you the output of the snippet:

```
$ python reduce_memory_consumption.py
[With __slots__] Total allocated size:
   ↪ 6.9 MB
[Without __slots__] Total allocated size
   ↪ : 16.8 MB
```

Listing 1.76: Output of reduce_memory_consumption.py

1.13.9 Reduce Memory Consumption - Using Iterator

You can reduce the memory consumption by using iterators whenever possible.

```
import sys

from itertools import repeat


lots_of_fours = repeat(4, times=100
   ↪ _000_000)
print(f"Using itertools.repeat: {sys.
   ↪ getsizeof(lots_of_fours)} bytes")

lots_of_fours = [4] * 100_000_000
print(
    f"Using list with 100.000.000
       ↪ elements: {sys.getsizeof(
```

```
        ↪ lots_of_fours) / (1024**2)} MB
        ↪ "
)
```

Listing 1.77: reduce_memory_consumption_iterator.py

Just have a look at the resulting output, it speaks for itself.

```
$ python
   ↪ reduce_memory_consumption_iterator
   ↪ .py
Using itertools.repeat: 56 bytes
Using list with 100.000.000 elements:
   ↪ 762.9395141601562 MB
```

Listing 1.78: Output of reduce_memory_consumption_iterator.py

1.13.10 RegEx Parse Tree

Print the ReqEx parse tree using `re.DEBUG`.

```
import re
```

```
re.compile("([\w\.-]+)@([\w\.-]+)", re.
    ↪ DEBUG)
```

Listing 1.79: regular_expression_debug.py

1.13.11 Scopes

The snippet `scopes_namespaces.py` contains an example to demonstrate the different scopes and namespaces available in Python. As it's only for demonstarting purposes and as the file is quite huge, it's not displayed here.

1.13.12 Set Union and Intersection

Python provides set union and intersection using the `&` and `|` operators.

```
set_a = {1, 2}
set_b = {2, 3}

print("Use | and & for set union and
    ↪ intersection.")
print(f"{set_a} & {set_b} = {set_a &
    ↪ set_b}")
```

```
print(f"{set_a} | {set_b} = {set_a |
    ↪ set_b}")
```

Listing 1.80: set_union_intersection.py

```
$ python set_union_intersection.py
{1, 2} & {2, 3} = {2}
{1, 2} | {2, 3} = {1, 2, 3}
```

Listing 1.81: set_union_intersection.py

1.13.13 Sort Complex Tuples

If you have a list of more complex tuples, you may want to sort them by a certain key. You can provide a key function for the builtin **sorted** function.

```
student_tuples = [("John", "A", 15), ("
    ↪ Jane", "B", 12), ("Dave", "B", 10)
    ↪ ]

sorted_list = sorted(student_tuples, key
    ↪ =lambda student: student[2])
```

```
print(sorted_list)
```

Listing 1.82: sort_complex_tuples.py

```
$ python sort_complex_tuples.py
[('Dave', 'B', 10), ('Jane', 'B', 12), (
    ↪ 'John', 'A', 15)]
```

Listing 1.83: Output of sort_complex_tuples.py

1.13.14 Unicode in Source Code

Python allows you to use unicode in your source, what you shouldn't do. Nevertheless, this snippet shows you a sample unicode usage in source code.

```
# Only works in Python REPL

def ^2(' ):
    f"result" = ' * '
    return f"result"


^2(4)
```

Listing 1.84: unicode_source_code.py

However, this only works in the REPL or in iPython, but not in files.

1.13.15 UUID

The `uuid` module provides methods to generade UUIDs. In this snippet you can find a sample implementation of UUID1.

```
from uuid import uuid1

for i in range(5):
    print(uuid1())
```

Listing 1.85: uuid1_example.py

The output may look like this:

```
$ python uuid1_example.py
3120c650-4e5e-11e9-be8f-dca904927157
3120ca1a-4e5e-11e9-be8f-dca904927157
3120cb0a-4e5e-11e9-be8f-dca904927157
```

```
3120cbbe-4e5e-11e9-be8f-dca904927157
3120cc72-4e5e-11e9-be8f-dca904927157
```

Listing 1.86: Output of uuid1_example.py

1.13.16 Zip Safe

This snippet illustrates how zip is stopping if one iterable is exhausted without a warning and how to prevent it.

```
import itertools


a = [1, 2, 3]
b = ["One", "Two"]

result1 = list(zip(a, b))
result2 = list(itertools.zip_longest(a,
    ↪ b))

print(result1)
print(result2)
```

Listing 1.87: zip_safe.py

```
$ python zip_safe.py
[(1, 'One'), (2, 'Two')]
[(1, 'One'), (2, 'Two'), (3, None)]
```

Listing 1.88: Output of zip_safe.py

1.13.17 Tree Clone

If you have ever worked with unix systems, you might know the `tree` command. The following Listing shows you a pure Python implementation of this command.

```
from pathlib import Path


def tree(directory):
    print(f"{directory}")
    for path in sorted(directory.rglob("
        ↪ *")):
        depth = len(path.relative_to(
            ↪ directory).parts)
        spacer = "    " * (depth - 1)
        print(f"{spacer}\u2514---- {path
            ↪ .name}")
```

```
tree(Path().cwd())
```

Listing 1.89: tree_clone.py

A sampl output might look like this:

```
$ python tree_clone.py
/Users/florian/workspace/python/test-
    ↪ directory
└──── app
      └──── __init__.py
      └──── __main__.py
└──── main.py
└──── test.py
└──── utils.py
```

Listing 1.90: Output of tree_clone.py

1.13.18 pathlib implementation of os.relpath

The following Listing is a `pathlib.Path` implementation of `os.relpath`, which should *never* be used in production. It's only meant to show a nice hacky implemen-

tation of `os.relpath` as its not provided by default by `pathlib.Path`.

```
from itertools import takewhile
from pathlib import Path


foo = Path("foo")
baz = Path("baz")

common_rel_path = Path(
    *[
        set(i).pop()
        for i in (takewhile(lambda x: x
           ↪ [0] == x[1], zip(foo.parts
           ↪ , baz.parts)))
    ]
)
print(common_rel_path)
```

Listing 1.91: pathlib_relpath.py

The snippet assumes that you have at least two directories in your current working directory: `foo` and `baz`. The output is shown in the following Listing:

```
$ python pathlib_relpath.py
.
```

Listing 1.92: Output of pathlib_relpath.py

1.13.19 Slicing Generators

Generators are a pretty nice thing. For instance you can write a function returning the fibonacci numbers using `yield` turning it into a generator, which is way more efficient than creating a function calculating a finite number of results and returning the resulting list. The problem: You're not able to slice the returned generator. Or at least you can't using the `[]`-operator. Making use of the awesome `itertools` module, you can!

```
import itertools


def fibonacci():
    current, nxt = 0, 1
    while True:
        current, nxt = nxt, nxt +
            ↪ current
        yield current
```

```
first_fove = list(itertools.islice(
    ↪ fibonacci(), 5))
print(f"First five: {first_fove}")
```

Listing 1.93: slice_generators.py

1.13.20 Count Elements Between Thresholds

If you want to get the number of elements between certain thresholds, you can make use of the following Listing.

```
data = [1, 2, 3, 4, 5, 6, 7]
low, high = 3, 6
result = sum(low <= x < high for x in
    ↪ data)

print(result)
```

Listing 1.94: count_thresholds.py

The essential point is, that you can sum booleans as they are a subclass of integers.

```
$ python count_thresholds.py
3
```

Listing 1.95: Output of count_thresholds.py

Chapter 2

Third Party

What distinguishs the recipes in this chapter from the ones in the *Standard Library* chapter? The ones in this chapter make use of third party packages you need to install first. You can find the used code snippets in the *third_party* directory of the repository.

2.1 Image and Animation

The section *Image and Animation* contains recipes for image and video manipulation as well as animation creation.

2.1.1 Create A GIF

You can create a GIF by using the following three third party packages:

- `animatplot`
- `matplotlib`
- `numpy`

```
import animatplot as amp
import matplotlib.pyplot as plt
import numpy as np


x = np.linspace(0, 1, 50)
t = np.linspace(0, 1, 20)

X, T = np.meshgrid(x, t)
Y = np.sin(2 * np.pi * (X + T))

block = amp.blocks.Line(X, Y)
anim = amp.Animation([block])

anim.controls()  # creates a
   ↪ timeline_slider and a play/pause
   ↪ toggle
```

```
anim.save_gif("images/line2")  # save
    ↪ animation for docs
plt.show()
```

Listing 2.1: animated_graphics.py

Note: Make sure, that an `images` directory exists as it's not created automatically.

2.1.2 Change Image Background

To be able to use this recipe you not only need to have `numpy` installed but also have `OpenCV` installed on your system. There is an unofficial pre-built OpenCV packages for Python availabl on PyPI. You can install it via `pip`.

```
$ python -m pip install opencv-python
```

Listing 2.2: Install the unofficial pre-build OpenCV Python package from PyPI

```
import cv2
import numpy as np
```

```
img = cv2.imread(open(""))

for i in range(img.shape[0]):
    for j in range(img.shape[1]):
        if all(img[i][j] == np.array
           ↪ ([114, 89, 47])):
            img[i][j] = np.array([200,
               ↪ 0, 0])

cv2.imwrite("new.jpg", img)
```

Listing 2.3: change_bg.py

Note: `np.array([114, 89, 47])` represents the background image you want to replace by the one on the next line.

2.1.3 Manipulate Images Using Imageio

`imageio` is a Python library that provides an easy interface to read and write a wide range of image data, including animated images, volumetric data, and scientific formats. The following Listing shows you how to read an image from an url, turn it into a grey one and finally save it after blurring.

```
import imageio
import numpy as np
import scipy.ndimage


start_img = imageio.imread(
    "http://static.cricinfo.com/db/
       ↪ PICTURES/CMS/263600/263697.20.
       ↪ jpg"
)
gray_inv_img = 255 - np.dot(start_img
   ↪ [..., :3], [0.299, 0.587, 0.114])
blur_img = scipy.ndimage.filters.
   ↪ gaussian_filter(gray_inv_img,
   ↪ sigma=5)


def dodge(front, back):
    result = front * 255 / (255 - back)
    result[np.logical_or(result > 255,
       ↪ back == 255)] = 255
    return result.astype("uint8")


final_img = dodge(blur_img, gray_inv_img
   ↪ )
imageio.imwrite("final.jpg", final_img)
```

Listing 2.4: manipulate_images.py

2.1.4 Resize Images

With `OpenCV` it's possible to manipulate images. This includes resizing images as well. This recipe shows you how all `.jpg` images in the current working directory can be resized.

```
import cv2
import glob


images = glob.glob("*.jpg")

for image in images:
    img = cv2.imread(image, 0)
    re = cv2.resize(img, (100, 100))
    cv2.imshow("Hey", re)
    cv2.waitKey(500)
    cv2.destroyAllWindows()
    cv2.imwrite("resized_" + image, re)
```

Listing 2.5: resize_images.py

2.1.5 Hide Image Inside Another

Making use of the steganography technique (hiding information), you can hide a whole image in another one. This can be achieved by using the Python Imaging Library (PIL) or the active developed fork *Pillow*. The recipe contained by the `steganography.py` file provides you a CLI, which you can use to hide an image inside another.

2.1.6 Create Own Images

Using four different packages, you can create your own (random generated) image. The following Listing reveals you the necessary source code.

```
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np
from PIL import Image


randomInt = np.random.randint(256, size
    ↪ =(1000, 1000))
randomFloat = tf.to_float(randomInt)

with tf.Session() as session:
```

```
    result = session.run(randomFloat)
    im = Image.fromarray(result, "RGB")

    plt.imshow(im)
    plt.show()
```

Listing 2.6: tensorflow_image.py

2.2 Command-Line

In this section you will find useful snippets when dealing with command-lines.

2.2.1 Attention Please!

Sometimes you need to create an eye catcher on the command-line. Therefore, you can use `termcolor` to create one. In the following code snippet you will find the code necessary to create a blinking white message on a red background.

```
import termcolor
```

```
termcolor.cprint("Hey, this is important
   ↪ !!!", "white", "on_red", attrs=["
   ↪ blink"])
```

Listing 2.7: attention_message.py

2.2.2 Colored Python

As indicated in the previous recipe, you can use `termcolor` ↪ to print colored strings on the command-line.

```
from termcolor import colored, cprint

# Normal print
print("Hello World!")

# Normal print with colored text
text = colored("Hello Red World!", "red"
   ↪ )
print(text)

# Termcolor colored print
cprint("Hello Blue World!", "blue")
```

Listing 2.8: colored_python.py

2.2.3 Generate CLI Help Strings

Making use of `docopt` you can generate cli help strings based on Python docstrings.

```
"""CLI HELP STRINGS
Usage:
    cli_help_strings.py
    cli_help_strings.py <name>
    cli_help_strings.py -h/--help
    cli_help_strings.py -v/--version
Options:
    <name>  Optional name argument.
    -h --help  Show this screen.
    -v --version  Show version.
"""
from docopt import docopt


def say_hello(name):
    return f"Hello {name}!"


if __name__ == "__main__":
    arguments = docopt(__doc__, version=
        ↪ "DEMO 1.0")
    if arguments["<name>"]:
```

```
        print(say_hello(arguments["<name
            ↪ >"]))
    else:
        print(arguments)
```

Listing 2.9: cli_help_strings.py

```
$ python cli_help_strings.py --help
CLI HELP STRINGS
Usage:
    cli_help_strings.py
    cli_help_strings.py <name>
    cli_help_strings.py -h|--help
    cli_help_strings.py -v|--version
Options:
    <name>  Optional name argument.
    -h --help  Show this screen.
    -v --version  Show version.
```

Listing 2.10: Output of cli_help_string.py

2.2.4 Parse And Print Colored Arguments

`clint` provides you the functionalities to parse command-line arguments and to print them in a colored way.

```
import os
import sys

from clint.arguments import Args
from clint.textui import puts, colored,
   ↪ indent

sys.path.insert(0, os.path.abspath("..")
   ↪ )

args = Args()

with indent(4, quote=">>>"):
    puts(colored.blue("Arguments passed
       ↪ in: ") + str(args.all))
    puts(colored.blue("Flags detected: "
       ↪ ) + str(args.flags))
    puts(colored.blue("Files detected: "
       ↪ ) + str(args.files))
    puts(colored.blue("NOT Files
       ↪ detected: ") + str(args.
```

```
        ↪ not_files))
    puts(colored.blue("Grouped Arguments
        ↪ : ") + str(dict(args.grouped))
        ↪ )

print()
```

Listing 2.11: clint_cli_tool.py

```
$ python clint_cli_tool.py
>>> Arguments passed in: []
>>> Flags detected: <args []>
>>> Files detected: []
>>> NOT Files detected: <args []>
>>> Grouped Arguments: {'_': <args []>}
```

Listing 2.12: Output of clint_cli_tool.py

Note: The keywords between >>> and the colon are blue.

2.2.5 Print Tables

You can use `prettytable` to print a table on the command-line.

```
from prettytable import PrettyTable


table = PrettyTable(["food", "price"])
table.add_row(["ham", "$2"])
table.add_row(["eggs", "$1"])
table.add_row(["spam", "$4"])

print(table)
```

Listing 2.13: display_tables.py

```
$ python display_tables.py
+-------+-------+
| food  | price |
+-------+-------+
| ham   |   $2  |
| eggs  |   $1  |
| spam  |   $4  |
+-------+-------+
```

Listing 2.14: Output of display_tables.py

2.2.6 Fancy CLI Header

Ever wondered how other projects create those fancy cli headers? You can use `pyfiglet` to achieve exactly that!

```
from pyfiglet import Figlet

f = Figlet(font="slant")
print(f.renderText("text to render"))
```

Listing 2.15: fancy_cli_header.py

The following output is not included as image. It's copied and pasted directly from the command-line. So if you recognize a non-fancy header, make sure to try it yourself to get a better impression of what you can achieve with this recipe.

```
    __                __       __
   / /_____  _  __/ /_    / /_____
  / __/ _ \| |/_/ __/   / __/ __ \
 / /_/  __/>  </ /_    / /_/ /_/ /
 \__/\___/_/|_|\__/    \__/\____/
```

Listing 2.16: Output of fancy_cli_header.py

2.2.7 Interactive CLI

By using `PyInquirer` you can create interactive command-line interfaces. As the file is to long to be displayed here, I refer to the `interactive_cli.py` file contained by the *third_party* directory of the repository. I recommend to run the snippet and get a feeling for what's possible with this amazing package.

2.2.8 Creating A Progress Bar

As the name already suggests, `progressbar` can be used to create a progress bar on the command-line.

```
from time import sleep
from progressbar import ProgressBar

bar = ProgressBar()
for i in bar(range(50)):
    sleep(0.5)
```

Listing 2.17: shows_progress.py

2.2.9 Creating A Simple Progress Bar

This code snippet is similar to the last one except the fact, that this one uses the `tqdm` package.

```
from time import sleep
from tqdm import tqdm


for i in tqdm(range(100)):
    sleep(0.2)
```

Listing 2.18: simple_progressbar.py

2.2.10 Cooked Input

„`cooked_input` is a Python package for getting, cleaning, converting, and validating command line input. If you think of input (raw_input in legacy Python) as raw input, then this is cooked input.“

```
import cooked_input as ci


cap_cleaner = ci.CapitalizationCleaner(
   ↪ style=ci.ALL_WORDS_CAP_STYLE)
```

```
name = ci.get_string(prompt="What is
    ↪ your name?", cleaners=[cap_cleaner
    ↪ ])

print(name)
```

Listing 2.19: cooked_input_example.py

```
$ python cooked_input_example.py
What is your name?: FlOriAn dAhliTz
Florian Dahlitz
```

Listing 2.20: Output of cooked_input_example.py

2.2.11 Image To ASCII Art

Once in a while you may want to enhance the look and feel of your command-line application by adding some ascii art to it. Luckily, the folowing to recipes allow you to create ascii arts based on images. As both recipes are quite long, they are not display here, but you can find them in the corresponding repository.

The first one saves the ascii art to a file called `out.txt` in your current working directory. The second

one prints the resulting ascii art directly to stdout. Make sure to checkout the code of the first recipe as it shows you, which arguments you can pass to the file.

```
$ python img_to_ascii.py --file src/
    ↪ python.png --cols 120
$ python img_to_ascii_2.py src/python.
    ↪ png
```

Listing 2.21: Usage of both recipes

2.3 Timing

The *Timing* section contains snippets related to timing. This means, that you'll find ways to time the execution time of certain code parts and much more. To achieve that, `boxx` is mainly used.

2.3.1 Timing Tool: boxx

The third party `boxx` module provides you a pretty neat way to time your code executions via `with`-blocks.

```
from boxx import timeit
```

```
from time import sleep


with timeit():
    sleep(0.01)

with timeit(name="sleep"):
    sleep(0.1)
```

Listing 2.22: timing_tool.py

```
$ python timing_tool.py
"timeit" spend time: 0.01047492
"sleep" spend time: 0.10027
```

Listing 2.23: Output of timing_tool.py

2.3.2 f-strings VS str

Let's say you want to convert an integer to a string. There exist different ways to do that. Working on an open source project I came across an option I've never thought of: using f-strings. But which method is faster? Using the builtin **str** function or f-strings? You can use `boxx` to time it easily.

```
from boxx import timeit


with timeit(name="f-strings"):
    for i in range(500_000):
        x = 6
        f"{x}"

with timeit(name="str"):
    for i in range(500_000):
        x = 6
        str(x)
```

Listing 2.24: f-strings_vs_str.py

```
$ python f-strings_vs_str.py
"f-strings" spend time: 0.09714508
"str" spend time: 0.1627851
```

Listing 2.25: Output of f-strings_vs_str.py

Note: Even though f-strings are faster in this situation, keep in mind, that the builtin **str** function is preferred to be used. Clean code is usually more important than efficiency.

2.3.3 range VS repeat

`itertools.repeat` is faster than **`range`** for looping a fixed number of times when you don't need the loop variable. An example is given below.

```
import itertools
import random

from boxx import timeit

with timeit(name="Range"):
    min(random.random() for i in range
       ↪ (100_000))

with timeit(name="Repeat"):
    min(random.random() for _ in
       ↪ itertools.repeat(None, 100_000
       ↪ ))
```

Listing 2.26: range_vs_repeat.py

As you can see, `itertools.repeat` is slightly faster.

```
$ python range_vs_repeat.py
"Range" spend time: 0.01480103
```

```
"Repeat" spend time: 0.01320004
```

Listing 2.27: Output of range_vs_repeat.py

If you want to avoid passing the `None` argument, you can create a new function called `times`, which hides the `None` argument.

```
from functools import partial
from itertools import repeat

times = partial(repeat, None)
min(random.random() for _ in times(100
    ↪ _000))
```

Listing 2.28: partial_function.py

2.4 Non-Categorized

All recipes, which do not fall into one of the underlying categories, are listed here.

2.4.1 Auto Login On Website

Using `selenium` allows you to automatically open a new browser window and login into a certain website, e.g. GitHub.

```
from selenium import webdriver


website_link = "https://github.com/login
    ↪ "
username = ""
password = ""

element_for_username = "login"
element_for_password = "password"
element_for_submit = "commit"

browser = webdriver.Firefox()
browser.get(website_link)

try:
    username_field = browser.
        ↪ find_element_by_name(
        ↪ element_for_username)
    username_field.send_keys(username)
    password_field = browser.
```

```
        ↪ find_element_by_name(
        ↪ element_for_password)
    password_field.send_keys(password)
    signInButton = browser.
        ↪ find_element_by_name(
        ↪ element_for_submit)
    signInButton.click()
except Exception as e:
    print(e)
```

Listing 2.29: auto_login_website.py

2.4.2 Count Python Bytes In A Directory

`PyFilesystem2` is Python's file system abstraction layer. You can use a file system object to analyse your files. The folowing recipe shows you how you can get the number of Python source code bytes in your directory.

```
import sys

from fs import open_fs
from fs.filesize import traditional
```

```
fs_url = sys.argv[1]
count = 0

with open_fs(fs_url) as fs:
    for _path, info in fs.walk.info(
        ↪ filter=["*.py"], namespaces=["
        ↪ details"]):
          count += info.size

print(f'There is {traditional(count)} of
   ↪  Python in "{fs_url}"')
```

Listing 2.30: count_python_bytes.py

2.4.3 Create World Maps

`folium` builds on the data wrangling strengths of the Python ecosystem and the mapping strengths of the Leaflet.js library. Manipulate your data in Python, then visualize it in a Leaflet map via `folium`. This recipe creates a world map with the USA in the centre of the map.

```
import folium

mapOBJ = folium.Map(location=[39.5,
   ↪ -98.35], zoom_start=4, tiles="
```

```
   ↪ Stamen Terrain")
mapOBJ.save(outfile="map.html")
```

Listing 2.31: folium_snippet.py

2.4.4 Print Formatted JSON

Printing JSON object formatted is as easy as:

```
import json
import requests


def sample_request(url: str):
    resp = requests.get(url)

    return resp.json()


response = sample_request("https://
   ↪ jsonplaceholder.typicode.com/todos
   ↪ /1")
print(json.dumps(response, indent=4))
```

Listing 2.32: formatted_json.py

```
$ python formatted_json.py
{
    "userId": 1,
    "id": 1,
    "title": "delectus aut autem",
    "completed": false
}
```

Listing 2.33: Output of formatted_json.py

Note: The snippet is part of the third party part as it makes use of the `requests` library. The `json` module is part of Pythons standard library.

2.4.5 Inspect Docker

You can inspect running Docker containers and existing images using the `docker` module.

```
import docker


client = docker.from_env()

print(client.containers.run("ubuntu
    ↪ :16.04", "echo hello world"))
```

```
print(client.containers.list())
print(client.images.list())
```

Listing 2.34: inspect_docker.py

2.4.6 Is Holiday

The `holidays` module provides you an elegant and easy way to check, whether a given date is a holiday in the specified region.

```
from datetime import date

import holidays


today = date.today()
is_holiday = today in holidays.Germany()

print(is_holiday)
```

Listing 2.35: is_holiday.py

2.4.7 Web Scraping

The recipe in the `mathematicians.py` file shows you how you can scrape information from the internet. It makes use of the following third party packages:

- BeautifulSoup
- requests

The recipe returns you the most popular mathematicians. As it's to long, please have a look at it in the repository.

2.4.8 Interacting With The Medium API

Another recipe, which is to long but worth to mention, is the one contianed by the `medium.py` file. Running the file gives you the ability to interact with the Medium API via the command-line.

2.4.9 Mocking Requests

When writing tests for your application, you may come across the situation, where you have to mock requests. The following Listing shows you how you can do this by using the standard libraries `unittest.mock` module.

```
from unittest.mock import Mock

import requests


requests = Mock()


def get_holidays():
    r = requests.get("http://localhost/
        ↪ api/holidays")
    if r.status_code == 200:
        return r.json()
    return None


class TestCalendar:
    def log_request(self, url):
        # Log a fake request for test
            ↪ output purposes
        print(f"Making a request to {url
            ↪ }.")
        print("Request received!")

        # Create a new Mock to imitate a
            ↪  Response
        response_mock = Mock()
```

```
        response_mock.status_code = 200
        response_mock.json.return_value
           ↪ = {
            "12/25": "Christmas",
            "7/4": "Independence Day",
        }

        return response_mock

    def test_get_holidays_logging(self):
        requests.get.side_effect = self.
           ↪ log_request
        print(requests)
        assert get_holidays()["12/25"]
           ↪ == "Christmas"
```

Listing 2.36: mocking_requests.py

2.4.10 Mypy Example

The following Listing reveals the usage of `mypy` as a static type checker. Running the snippet will throw a `TypeError` as expected.

```
import mypy
```

```
def add(a: int, b: int) -> int:
    return a + b


print(add(1, 5))
print(add("a", 3))
```

Listing 2.37: mypy_example.py

2.4.11 NumPy Array Operations

Running the following code snippet reveals you some of the existing `numpy` array operations.

```
import numpy as np


x = np.array([1, 2, 3, 4, 5])

print(f"{x * 2}")
print(f"{x * x}")
```

Listing 2.38: numpy_array_operations.py

```
$ python numpy_array_operations.py
[ 2  4  6  8 10]
[ 1  4  9 16 25]
```

Listing 2.39: Output of numpy_array_operations.py

2.4.12 Parse Complex Excel Sheets

You can parse more complex Excel Sheets by using `pandas` ↪ and installed a `pandas` extension called `xlrd`.

```
from pathlib import Path

import pandas as pd


path = Path("src/samples.xlsx")
xls = pd.ExcelFile(path)
print(xls.sheet_names)

df = xls.parse(xls.sheet_names[0])
print(df.head())
```

Listing 2.40: parse_complex_excel_sheets.py

2.4.13 Nmap For Python

Creating your own port scanner becomes easy when using `nmap`. Luckily, a Python package exists providing access to nmap: `python-nmap`. **Hint:** Make sure you have Nmap installed on your operating system as `python-nmap` only provides access to the Nmap API.

```
import nmap

nm = nmap.PortScanner()
nm.scan("127.0.0.1", "22-443")

for host in nm.all_hosts():
    print("
       ↪ ----------------------------------------------------
       ↪ ")
    print(f"Host : {host} ({nm[host].
       ↪ hostname()})")
    print(f"State : {nm[host].state()}")

    for protocol in nm[host].
       ↪ all_protocols():
        print("----------")
        print(f"Protocol : {protocol}")
        port_list = sorted(nm[host][
           ↪ protocol].keys())
```

```
        for port in port_list:
            print(f"port : {port}\tstate
               ↪  : {nm[host][protocol
               ↪ ][port]['state']}")
```

Listing 2.41: port_scanner_nmap.py

2.4.14 Test Renamed Class

Renaming a class can happen over time. If you want to test, wether your code is backwards compatible, you can make use of the following snippet.

```
import pytest


class Mario:
    pass


class Luigi:
    pass


@pytest.fixture(params=[Mario, Luigi])
```

```
def plumber(request):
    plumber_cls = request.param

    return plumber_cls()


def test_backwards_compatibility(plumber
    ↪ ):
    assert 1 == 1
```

Listing 2.42: pytest_rename_class_backwards_compatibility.py

2.4.15 Reduce Pandas Dataframe Memory

When dealing with large data sets, it can be an advantage to change column types from `object` to `category` as shown in the next Listing.

```
"""The used data set can be found here:
    ↪ https://www.kaggle.com/worldbank/
    ↪ world-development-indicators#
    ↪ Indicators.csv"""
import pandas as pd
```

```
df = pd.read_csv("data/indicators.csv")
print("========== Before (with object)
   ↪ ==========")
print(df.info())

for column in df.select_dtypes("object")
   ↪ :
    df[column] = df[column].astype("
       ↪ category")

print("========== After (with category)
   ↪ ==========")
print(df.info())
```

Listing 2.43: reduce_pandas_df_memory.py

2.4.16 Async Libraries

Before `async`/`await` became so popular and part of the standard library, you had several options to use asynchronous techniques in your project. The three main ones were `asynio`, `gevent` and `tornado`. You can find an example for each in the *third_party* directory of the repository. The file names are as follows:

- `test_asyncio.py`

- `test_gevent.py`
- `test_tornado.py`

2.4.17 Unzip World Bank Data

The following recipe shows you how you can download a `.zip` file from an online source, extract and work with the data. Therefore, except the `requests` library only standard library modules are used.

```
import csv
import io
import zipfile

import requests


url = "http://databank.worldbank.org/
    ↪ data/download/WDI_csv.zip"
filename = "WDIData.csv"

# Warning: this can take two minutes to
    ↪ download!!
with zipfile.ZipFile(io.BytesIO(requests
    ↪ .get(url).content)) as zip_file:
```

```
    print("\n".join(name for name in
       ↪ zip_file.namelist()))
    zip_file.extractall()

# On Python 2, remove: ", newline=''"
with open(filename, newline="") as
   ↪ in_file:
    for row in csv.reader(in_file):
        print(", ".join(row))
```

Listing 2.44: world_bank_data.py

2.4.18 Simple Debugger

When writing Python code you may come across situations, where you want to find out, what's going on. Sometimes you may make use of an actual debugger, but most of the times inserting some **print**-statements seems just fine. The `PySnooper` package is probably the best reason, why you should never use **print**-statements to debug your code again. The following Listing shows you the simple usage of its `snoop` decorator.

```
import pysnooper
```

```
@pysnooper.snoop()
def foo():
    a = [x for x in range(100)]
    return [x for x in a if x % 2 == 0]


foo()
```

Listing 2.45: simple_debugger.py

The output is too long to be covered here. Feel free to run the snippet from your terminal and get a sense of how amazing and helpful this little package can be.

2.4.19 Text Analysis

If you want a simple way to perform text analysis, you can use `TextBlob`. The following snippet shows you a sample usage. Make sure to read the docs of the package as it provides functionality for translations and further analysis, too.

```
from pathlib import Path

from textblob import TextBlob
```

```
path = Path("src/text.txt")

with open(path) as f:
    text = f.read()

blob = TextBlob(text)

for sentence in blob.sentences:
    print(sentence.sentiment.polarity)
```

Listing 2.46: text_analysis.py

Note: Make sure to download the needed data before hand, otherwise an exception is thrown telling you to download it.

```
>>> import nltk
>>> nltk.download('
   ↪ averaged_perceptron_tagger')
```

Listing 2.47: Download data using NLTK

2.4.20 Complex List Ops

In your day to day work you will very likely come across performance issues in Python. Sometimes it's a good

idea to choose a different data structure than you have used to speed up certain operations. However, you may come across situations where this is even not enough. Instead of switching to another language like C, you may want to check out the `blist` package first. „The `blist` is a drop-in replacement for the Python list that provides better performance when modifying large lists.“Here is a small snippet illustrating the huge impact of using a `blist` instead of a **`list`**:

```
from blist import blist
from boxx import timeit


def complex_list_ops(x: list) -> list:
    x *= 2**29
    x.append(5)
    y = x[4:-234234]
    del x[3:1024]
    return x


with timeit(name="Builtin"):
    y = complex_list_ops([0])

with timeit(name="Blist"):
    y = complex_list_ops(blist([0]))
```

Listing 2.48: complex_list_ops.py

```
$ python complex_list_ops.py
"Builtin" spend time: 24.61005
"Blist" spend time: 4.675813
```

Listing 2.49: Output of complex_list_ops.py

„The blist package also provides `sortedlist`, `sortedset` ↪ , [...] `btuple` types“, and much more.