Python Snippets

A Collection of Useful Python Snippets with Explanations

Florian Dahlitz

March 27, 2019

Contents

Contents **2**

1 Standard Library **5**
1.1 Abstract Base Class (ABC) 5
1.1.1 General Usage 5
1.2 AsyncIO 7
1.2.1 Time Saving 7
1.3 Playing with Bits 11
1.3.1 Bit Flipper 11
1.3.2 A Simple Bitmask 13
1.4 Search Algorithms 15
1.4.1 Binary Search 15
1.5 Dealing with builtins 17
1.5.1 Change Behaviour 17
1.5.2 Capture Output 19
1.6 String Operations 20
1.6.1 Check for String Pattern 20
1.6.2 Compare Strings 22

1.6.3 Fill Strings 23
1.6.4 Parse Query String 24
1.6.5 Print Numbers Human-Friendly . . 25
1.6.6 Split Strings Preserving Substrings 26
1.6.7 capitalize() vs. title() 27
1.7 Dictionaries 28
1.7.1 Crazy Dictionary Expression . . . 28
1.7.2 Emulate switch-case 29
1.7.3 Merge Abitrary Number of Dicts . 30
1.7.4 MultiDict with Default Values . . . 31
1.7.5 Overwrite Dictionary Values 33
1.7.6 Pass Multiple Dicts as Argument . 34
1.7.7 Default Configuration 35
1.7.8 Keep Original After Updates . . . 37
1.7.9 Update Dict Using Tuples 38
1.7.10 Create A Dict Based On Lists . . . 39
1.8 Decorators 41
1.8.1 Deprecation Decorator 41
1.8.2 Keep Function Metadata 44
1.8.3 Trace Decorator 46
1.9 Bytecode 48
1.9.1 Disassemble Bytecode - String Conversion 48
1.9.2 Human Readable Bytecode 50
1.10 Lists 51
1.10.1 Flatten a List 51
1.10.2 Priority Queue 53
1.10.3 Remove Duplicates 55

1.10.4 Unpacking Lists Using * Operator 56
1.11 Files 57
1.11.1 Hash a File 57
1.11.2 Read Files using Iterator 59
1.12 Context Manager 59
1.12.1 Open Multiple Files 59
1.12.2 Temporal SQLite Table 60
1.13 Non-Categorized 62
1.13.1 Turtle 63
1.13.2 Function Parameters 64
1.13.3 Password Input 65
1.13.4 Hex Decode 66
1.13.5 MicroWebServer 66
1.13.6 Open Browser Tab 67
1.13.7 Port Scanner 67
1.13.8 Reduce Memory Consumption - Customizing ___slots___ 68
1.13.9 Reduce Memory Consumption - Using Iterator 68
1.13.10 RegEx Parse Tree 70
1.13.11 Scopes 70
1.13.12 Set Union and Intersection 71
1.13.13 Sort Complex Tuples 72
1.13.14 Unicode in Source Code 73
1.13.15 UUID 73
1.13.16 Zip Safe 74
1.13.17 Tree Clone 75

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

1.11 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 1.61: 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 1.62: read_files_using_iterator.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 1.63: 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 1.64: 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 1.65: Output of temptable_contextmanager.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 1.66: 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 1.67: 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 1.68: 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 1.69: hex_decode.py

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

Listing 1.70: 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 1.71: 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 1.72: 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 1.73: 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 1.74: 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 1.75: 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 1.76: set_union_intersection.py

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

Listing 1.77: 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 1.78: sort_complex_tuples.py

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

Listing 1.79: 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 1.80: 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 1.81: 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 1.82: 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 1.83: zip_safe.py

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

Listing 1.84: 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 1.85: 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 1.86: Output of tree_clone.py