Python-Snippets Documentation

Release 0.1

Shipeng Feng

Nov 01, 2017

Contents

Python-Snippets is a basket of Python snippets. Some of them are from Stack Overflow, others are written by me or taken from various web resources.

CHAPTER 1

Basic

Snippets on basic things.

1.1 not None test

```
if value is not None:
    pass
```

1.2 dump variable

```
import pprint

pprint.pprint(globals())
```

1.3 ternary conditional operator

```
>>> 'true' if True else 'false'
'true'
>>> 'true' if False else 'false'
'false'
```

1.4 accessing the index in for loops

```
for idx, val in enumerate(ints):
    print idx, val
```

1.5 how do I check if a variable exists

To check the existence of a local variable:

```
if 'myVar' in locals():
    # myVar exists.
```

To check the existence of a global variable:

```
if 'myVar' in globals():
    # myVar exists.
```

To check if an object has an attribute:

```
if hasattr(obj, 'attr_name'):
    # obj.attr_name exists.
```

CHAPTER 2

String

Snippets on string manipulation.

2.1 check if a string is a number

```
>>> a = "123"
>>> a.isdigit()
True
>>> b = "123abc"
>>> b.isdigit()
False
```

2.2 reverse a string

```
>>> 'hello world'[::-1]
'dlrow olleh'
```

2.3 write long string

If you follow PEP8, there is a maximum of 79 characters limit, BTW, you should follow PEP8, :)

```
string = ("this is a "
          "really long long "
          "string")
```

2.4 number with leading zeros

```
>>> "%02d" % 1
'01'
>>> "%02d" % 100
'100'
>>> str(1).zfill(2)
'01'
>>> str(100).zfill(2)
'100'
>>> int('01')
1
```

CHAPTER 3

List

Snippets on list manipulation.

3.1 check if a list is empty

```
if not a:
    print "List is empty"
```

3.2 empty a list

```
del l[:]
```

3.3 shuffle a list

```
from random import shuffle
# works in place
l = range(10)
shuffle(l)
```

3.4 append vs extend

append:

```
x = [1, 2, 3]
x.append([4, 5])
# [1, 2, 3, [4, 5]]
```

extend:

```
x = [1, 2, 3]
x.extend([4, 5])
# [1, 2, 3, 4, 5]
```

3.5 split a list into evenly sized chunks

Here's a generator that yields the chunks you want:

```
def chunks(l, n):
    """ Yield successive n-sized chunks from l.
    """
    for i in xrange(0, len(l), n):
        yield l[i:i+n]
```

3.6 finding the index of an item

```
>>> ["foo","bar","baz"].index('bar')
1
```

3.7 sort a list of dictionaries by values of the dictionary

```
newlist = sorted(list_to_be_sorted, key=lambda k: k['name'])
```

3.8 randomly select an item from a list

```
import random

foo = ['a', 'b', 'c', 'd', 'e']
random.choice(foo)
```

3.9 list comprehension

```
new_list = [i * 2 for i in [1, 2, 3]]
```

CHAPTER 4

Dictionary

Snippets on dict manipulation.

4.1 merge (union) two Python dictionaries

```
>>> x = {'a':1, 'b': 2}
>>> y = {'b':10, 'c': 11}
>>> z = x.copy()
>>> z.update(y)
{'a': 1, 'c': 11, 'b': 10}
```

4.2 create a dictionary with list comprehension

In Python 2.6 (or earlier), use the dict constructor:

```
d = dict((key, value) for (key, value) in sequence)
```

In Python 2.7+ or 3, you can just use the dict comprehension syntax directly:

```
d = {key: value for (key, value) in sequence}
```

CHAPTER 5

Class

Snippets on classes.

5.1 New-style and classic classes

Classes and instances come in two flavors: old-style and new-style. For more details, check out New-style and classic classes

5.2 print a object

```
Class User(object):

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

    def __repr__(self):
        return '<User %r>' % self.name

    def __str__(self):
        return self.name
```

CHAPTER 6

Decorator

Snippets about decorators.

6.1 decorators 101

Let's begin with syntactic sugar:

```
@EXPR
def add(a, b):
    return a + b

# The same thing as

def add(a, b):
    return a + b
add = EXPR(add)

@EXPR(ARG)
def add(a, b):
    return a + b

# The same thing as

def add(a, b):
    return a + b
add = EXPR(ARG)(add)
```

6.2 good decorator

We want to run a function for certain times, if you do not know what does `functools.wraps` do, check out What does functools.wraps do:

```
from functools import wraps

def spam(repeats):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            for i in range(repeats):
                func(*args, **kwargs)
        return wrapper
    return decorator

@spam(11)
def talk(word):
    """talk docstring"""
    print word
```

6.3 method decorator

We need one decorator that makes function to method decorators:

```
from functools import update_wrapper

class MethodDecoratorDescriptor(object):

    def __init__(self, func, decorator):
        self.func = func
        self.decorator = decorator

    def __get__(self, obj, type=None):
        return self.decorator(self.func.__get__(obj, type))

def method_decorator(decorator):
    def decorate(f):
        return MethodDecoratorDescriptor(f, decorator)
    return decorate

# usage
def spam(f):
    def wrapper(value):
        return f(value) + ":spamed"
    return update_wrapper(wrapper, f)

class MyClass(object):

    @method_decorator(spam)
    def my(self, value):
        return value

foo = MyClass()
print foo.my('hello')
```

CHAPTER 7

Iterator And Generator

Snippets about iterators and generators.

7.1 iterator - the Python yield keyword explained

Iterables:

```
>>> mylist = [x*x for x in range(3)]
>>> for i in mylist:
...     print(i)
0
1
4
```

Generators are iterators, but you can only iterate over them once. It's because they do not store all the values in memory, they generate the values on the fly:

```
>>> mygenerator = (x*x for x in range(3))
>>> for i in mygenerator:
...     print(i)
0
1
4
```

Yield is a keyword that is used like return, except the function will return a generator:

```
>>> def createGenerator():
...     mylist = range(3)
...     for i in mylist:
...         yield i*i
...
>>> mygenerator = createGenerator() # create a generator
>>> print(mygenerator) # mygenerator is an object!
<generator object createGenerator at 0xb7555c34>
```

```
>>> for i in mygenerator:
...     print(i)
0
1
4
```

7.2 how do I determine if an object is iterable

Duck typing:

```
try:
    iterator = iter(theElement)
except TypeError:
    # not iterable
else:
    # iterable
```

Type checking, need at least Python 2.6 and work only for new-style classes:

```
import collections

if isinstance(theElement, collections.Iterable):
    # iterable
else:
    # not iterable
```

CHAPTER 8

Descriptor

Snippets about descriptors.

8.1 descriptors 101

What is descriptor:

- __get__
- __set__
- __delete__
- Common descriptors: function, property

Demo:

```
>>> class Demo(object):
...     def hello(self):
...         pass
...
>>> Demo.hello
<unbound method Demo.hello>
>>> Demo.__dict__['hello']
<function hello at 0x102b17668>
>>> Demo.__dict__['hello'].__get__(None, Demo)
<unbound method Demo.hello>
>>>
>>> Demo().hello
<bound method Demo.hello of <__main__.Demo object at 0x102b28550>>
>>> Demo.__dict__['hello'].__get__(Demo(), Demo)
<bound method Demo.hello of <__main__.Demo object at 0x102b285d0>>
```

8.2 data and non-data descriptors

If an object defines both `__get__` and `__set__`, it is considered a data descriptor. Descriptors that only define `__get__` are called non-data descriptors.

Data and non-data descriptors differ in how overrides are calculated with respect to entries in an instance's dictionary. If an instance's dictionary has an entry with the same name as a data descriptor, the data descriptor takes precedence. If an instance's dictionary has an entry with the same name as a non-data descriptor, the dictionary entry takes precedence.

Demo:

```
class Demo(object):

    def __init__(self):
        self.val = 'demo'

    def __get__(self, obj, objtype):
        return self.val


class Foo(object):

    o = Demo()
```

```
>>> f = Foo()
>>> f.o
'demo'
>>> f.__dict__['o'] = 'demo2'
>>> f.o
'demo2'
```

Now we add `__set__`:

```
class Demo(object):

    def __init__(self):
        self.val = 'demo'

    def __get__(self, obj, objtype):
        return self.val

    def __set__(self, obj, val):
        self.val = val
```

```
>>> f = Foo()
>>> f.o
'demo'
>>> f.__dict__['o'] = 'demo2'
>>> f.o
'demo'
>>> f.o = 'demo3'
>>> f.o
'demo3'
```

8.3 better way to write property

Let's just see the code:

```
class Foo(object):

    def _get_thing(self):
        """Docstring"""
        return self._thing

    def _set_thing(self, value):
        self._thing = value

    thing = property(_get_thing, _set_thing)
    del _get_thing, _set_thing
```

8.4 cached property

Just grab it from django source:

```
class cached_property(object):
    """
    Decorator that creates converts a method with a single
    self argument into a property cached on the instance.
    """
    def __init__(self, func):
        self.func = func

    def __get__(self, instance, type):
        res = instance.__dict__[self.func.__name__] = self.func(instance)
        return res
```

CHAPTER 9

Context Manager

Snippets about with blocks.

9.1 open and close files

```
with open('data.txt') as f:
    data = f.read()
```

9.2 threading locks

```
lock = threading.Lock()

with lock:
    print 'do something'
```

9.3 ignore exceptions

```
from contextlib import contextmanager

@contextmanager
def ignored(*exceptions):
    try:
        yield
    except exceptions:
        pass

with ignored(ValueError):
    int('string')
```

9.4 assert raises

This one is taken from Flask source:

```
from unittest import TestCase

class MyTestCase(TestCase):
    def assert_raises(self, exc_type):
        return _ExceptionCatcher(self, exc_type)

class _ExceptionCatcher(object):

    def __init__(self, test_case, exc_type):
        self.test_case = test_case
        self.exc_type = exc_type

    def __enter__(self):
        return self

    def __exit__(self, exc_type, exc_value, tb):
        exception_name = self.exc_type.__name__
        if exc_type is None:
            self.test_case.fail('Expected exception of type %r' %
                                exception_name)
        elif not issubclass(exc_type, self.exc_type):
            reraise(exc_type, exc_value, tb)
        return True

class DictTestCase(MyTestCase):

    def test_empty_dict_access(self):
        d = {}
        with self.assert_raises(KeyError):
            d[42]
```

CHAPTER 10

Method

Snippets about method.

10.1 method 101

Instancemethods take `self` argument, classmethods take `cls` argument, staticmethods take no magic argument(not very useful).

```
class FancyDict(dict):
    @classmethod
    def fromkeys(cls, keys, value=None):
        data = [(key, value) for key in keys]
        return cls(data)
```

```
>>> FancyDict(key1='value1', key2='value2', key3='value3')
{'key3': 'value3', 'key2': 'value2', 'key1': 'value1'}
>>> FancyDict.fromkeys(['key1', 'key2'], 'value')
{'key2': 'value', 'key1': 'value'}
```

We are gonna talk a little more about classes, `__slots__`, you can use it to omit dict of python instances and reduce memory use in the end:

```
class Tiny(object):
    __slots__ = ['value']
    def __init__(self, value):
        self.value = value
t = Tiny(1)
```

```
>>> t.value
1
>>> t.value2 = 2
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
AttributeError: 'Tiny' object has no attribute 'value2'
```

Then talk a little about __new__, this is the actual constructor method of one instance, and it is one static method:

```
class WrappingInt(int):
    __slots__ = []
    def __new__(cls, value):
        value = value % 255
        self = int.__new__(cls, value)
        return self

wrapping_int = WrappingInt(256)
```

```
>>> wrapping_int
1
```

CHAPTER 11

Metaclass

Snippets about metaclasses.

11.1 metaclasses 101

In Python, everything is one object. Take CPython for example, we have PyObject, everything else is inherited from it, and each PyObject has one type attribute, PyTypeObject, actually PyTypeObject is one PyObject by itself too, so PyTypeObject has one type attribute, PyTypeType, PyTypeType has type attribute too, which is PyTypeType itself. So come back to Python, we have object, each object has one type, and the type of `type` is `type`.

You can also have a look on this stack overflow question What is a metaclass in Python.

Ok, in Python, everything is simple, so you can just treat metaclass as the class that creates a class. This is useful for post-processing classes, and is capable of much more magic(Dangerous though):

```
class ManglingType(type):
    def __new__(cls, name, bases, attrs):
        for attr, value in attrs.items():
            if attr.startswith("__"):
                continue
            attrs["foo_" + attr] = value
            del attrs[attr]
        return type.__new__(cls, name, bases, attrs)

class MangledClass:
    __metaclass__ = ManglingType

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

    def test(self):
        return self.value

mangled = MangledClass('test')
```

```
>>> mangled.test()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'MangledClass' object has no attribute 'test'
>>> mangled.foo_test()
'test'
```

CHAPTER 12

Performance

Snippets on Python running performance.

12.1 why does Python code run faster in a function

You might ask why it is faster to store local variables than globals. This is a CPython implementation detail.

Remember that CPython is compiled to bytecode, which the interpreter runs. When a function is compiled, the local variables are stored in a fixed-size array (not a dict) and variable names are assigned to indexes. This is possible because you can't dynamically add local variables to a function. Then retrieving a local variable is literally a pointer lookup into the list and a refcount increase on the PyObject which is trivial.

Contrast this to a global lookup (`LOAD_GLOBAL`), which is a true dict search involving a hash and so on. Incidentally, this is why you need to specify global i if you want it to be global: if you ever assign to a variable inside a scope, the compiler will issue `STORE_FAST` for its access unless you tell it not to.

By the way, global lookups are still pretty optimised. Attribute lookups foo.bar are the really slow ones!

CHAPTER 13

Module

Snippets on packages and modules.

13.1 get all module files

```
def _iter_module_files():
    for module in sys.modules.values():
        filename = getattr(module, '__file__', None)
        if filename:
            if filename[-4:] in ('.pyc', '.pyo'):
                filename = filename[:-1]
            yield filename
```

13.2 dynamic module import

```
def import_string(import_name):
    """Import a module based on a string.

    :param import_name: the dotted name for the object to import.
    :return: imported object
    """
    if '.' in import_name:
        module, obj = import_name.rsplit('.', 1)
    else:
        return __import__(import_name)
    return getattr(__import__(module, None, None, [obj]), obj)
```

CHAPTER 14

Input and Ouput

Snippets about input and output.

14.1 read from stdin

```
import sys

for line in sys.stdin:
    print line
```

14.2 print to stderr

```
print >> sys.stderr, 'spam'
```

CHAPTER 15

URL

Snippets for url.

15.1 url quote

```
from urllib import quote_plus

quoted = quote_plus(url)
```

15.2 url encode

```
from urllib import urlencode

encoded = urlencode({'key': 'value'})
encoded = urlencode([('key1', 'value1'), ('key2', 'value2')])
```

CHAPTER 16

HTTP

Snippets for http.

16.1 simple http web server

```
$ python -m SimpleHTTPServer [port]
```

16.2 http get

```
import urllib2
urllib2.urlopen("http://example.com/").read()
```

16.3 http head

```
>>> import httplib
>>> conn = httplib.HTTPConnection("www.google.com")
>>> conn.request("HEAD", "/index.html")
>>> res = conn.getresponse()
>>> print res.status, res.reason
200 OK
```

16.4 http post

```
>>> import httplib, urllib
>>> params = urllib.urlencode({'key': 'value'})
>>> headers = {"Content-type": "application/x-www-form-urlencoded",
...            "Accept": "text/plain"}
>>> conn = httplib.HTTPConnection("www.example.com")
>>> conn.request("POST", "", params, headers)
>>> response = conn.getresponse()
>>> print response.status, response.reason
200 OK
>>> conn.close()
```

16.5 http put

```
import urllib2
opener = urllib2.build_opener(urllib2.HTTPHandler)
request = urllib2.Request('http://example.org', data='your_put_data')
request.add_header('Content-Type', 'your/contenttype')
request.get_method = lambda: 'PUT'
url = opener.open(request)
```

CHAPTER 17

File

Snippets about files and folders manipulation.

17.1 check if file exists

If you need to be sure it's a file:

```
import os.path
os.path.isfile(filename)
```

17.2 unicode (utf8) reading and writing to files

It is easier to use the open method from the codecs module:

```
>>> import codecs
>>> f = codecs.open("test", "r", "utf-8")
>>> f.read()
```

17.3 extracting extension from filename

```
import os
filename, ext = os.path.splitext('/path/to/somefile.ext')
```

17.4 get file creation & modification date/times

ctime() does not refer to creation time on **nix* systems, but rather the last time the inode data changed:

```
import os.path, time
print "last modified: %s" % time.ctime(os.path.getmtime(file))
print "created: %s" % time.ctime(os.path.getctime(file))
```

17.5 find current directory and file's directory

```
os.getcwd()
os.path.dirname(os.path.realpath(__file__))
```

17.6 remove/delete a folder that is not empty

```
import shutil

shutil.rmtree('/folder_name')
```

17.7 recursively walk a directory

```
import os
root = 'your root path here'

# dirs are the directory list under dirpath
# files are the file list under dirpath
for dirpath, dirs, files in os.walk(root):
    for filename in files:
        fullpath = os.path.join(dirpath, filename)
        print fullpath
```

17.8 get file size

```
>>> import os
>>> statinfo = os.stat('index.rst')
>>> statinfo.st_size
487
```

17.9 reading binary file

```
with open("myfile", "rb") as f:
    byte = f.read(1)
    while byte:
        # Do stuff with byte.
        byte = f.read(1)
```

CHAPTER 18

Email

Snippets about email.

18.1 sending email with python

I recommend that you use the standard packages email and smtplib together:

```
# Import smtplib for the actual sending function
import smtplib

# Import the email modules we'll need
from email.mime.text import MIMEText

# Create a text/plain message
msg = MIMEText(body)

# me == the sender's email address
# you == the recipient's email address
msg['Subject'] = 'The contents of %s' % textfile
msg['From'] = me
msg['To'] = you

# Send the message via our own SMTP server, but don't include the
# envelope header.
s = smtplib.SMTP('localhost')
s.sendmail(me, [you], msg.as_string())
s.quit()
```

For multiple destinations:

```
# Import smtplib for the actual sending function
import smtplib

# Here are the email package modules we'll need
```

```
from email.mime.image import MIMEImage
from email.mime.multipart import MIMEMultipart

COMMASPACE = ', '

# Create the container (outer) email message.
msg = MIMEMultipart()
msg['Subject'] = 'Our family reunion'
# me == the sender's email address
# family = the list of all recipients' email addresses
msg['From'] = me
msg['To'] = COMMASPACE.join(family)
msg.preamble = 'Our family reunion'

# Assume we know that the image files are all in PNG format
for file in pngfiles:
    # Open the files in binary mode.  Let the MIMEImage class automatically
    # guess the specific image type.
    fp = open(file, 'rb')
    img = MIMEImage(fp.read())
    fp.close()
    msg.attach(img)

# Send the email via our own SMTP server.
s = smtplib.SMTP('localhost')
s.sendmail(me, family, msg.as_string())
s.quit()
```

18.2 sending mail via sendmail from python

If I want to send mail not via SMTP, but rather via sendmail:

```
from email.mime.text import MIMEText
from subprocess import Popen, PIPE

msg = MIMEText("Here is the body of my message")
msg["From"] = "me@example.com"
msg["To"] = "you@example.com"
msg["Subject"] = "This is the subject."
p = Popen(["/usr/sbin/sendmail", "-t"], stdin=PIPE)
p.communicate(msg.as_string())
```

CHAPTER 19

Datetime

Snippets about datetime.

19.1 converting string into datetime

```
from datetime import datetime

date_object = datetime.strptime('Jun 1 2005  1:33PM', '%b %d %Y %I:%M%p')
```

CHAPTER 20

Subprocess

Snippets about subprocesses.

20.1 calling an external command

```
from subprocess import call
call(["ls", "-l"])
```

20.2 subprocess input, output and returncode

```
from subprocess import Popen, PIPE

p = Popen(["ls", "-l"], stdin=PIPE, stdout=PIPE, stderr=PIPE)
std_input = "/"
out, err = p.communicate(std_input)
returncode = p.returncode
```

20.3 use string to call subprocess instead of list

```
from subprocess import call
import shlex

command = "ls -l"
call(shlex.split(command))
```

CHAPTER 21

Contribute

Pull requests are welcomed, thank you for your suggestions!