Advanced Python — exercises and solutions

Solutions have been inserted between the original text of the exercises. Take care :)

Exercise D1 (30 min)

Write a decorator which wraps functions to log function arguments and the return value on each call. Provide support for both positional and named arguments (your wrapper function should take both `*args` and `**kwargs` and print them both):

```
>>> @logged
... def func(*args):
...     return 3 + len(args)
>>> func(4, 4, 4)
you called func(4, 4, 4)
it returned 6
6
```

Solution

As a class:

```
class logged:
    def __init__(self, func):
        self.func = func

    def __call__(self, *args, **kwargs):
        print('you called {.__name__}({}{}{})'.format(
             func,
             str(list(args))[1:-1], # cast to list is because tuple
                                    # of length one has an extra comma
             ', ' if kwargs else '',
             ', '.join('{}={}'.format(*pair) for pair in kwargs.items()),
             ))
        val = func(*args, **kwargs)
        print('it returned', val)
        return val
```

As a function:

```
def logged(func):
    """Print out the arguments before function call and
    after the call print out the returned value
    """

    def wrapper(*args, **kwargs):
        print('you called {.__name__}({}{}{})'.format(
```

```
        func,
        str(list(args))[1:-1], # cast to list is because tuple
                               # of length one has an extra comma
        ', ' if kwargs else '',
        ', '.join('{}={}'.format(*pair) for pair in kwargs.items()),
        ))
    val = func(*args, **kwargs)
    print('it returned', val)
    return val
return wrapper
```

Long version with doctests and improved introspection:

```
import functools

def logged(func):
    """Print out the arguments before function call and
    after the call print out the returned value

    >>> @logged
    ... def func(*args):
    ...     return 3 + len(args)
    >>> func(4, 4, 4)
    you called func(4, 4, 4)
    it returned 6
    6

    >>> @logged
    ... def func2(a=None, b=None):
    ...     return None
    >>> func2()
    you called func2()
    it returned None
    >>> func2(3, b=2)
    you called func2(3, b=2)
    it returned None

    >>> @logged
    ... def func3():
    ...     "this function is documented"
    ...     pass
    >>> print(func3.__doc__)
    this function is documented
    """

    def wrapper(*args, **kwargs):
        print('you called {.__name__}({}{}{})'.format(
             func,
             str(list(args))[1:-1], # cast to list is because tuple
                                    # of length one has an extra comma
             ', ' if kwargs else '',
             ', '.join('{}={}'.format(*pair) for pair in kwargs.items()),
             ))
        val = func(*args, **kwargs)
        print('it returned', val)
        return val
    return functools.update_wrapper(wrapper, func)
```

Exercise D2 (20 min)

Write a decorator to cache function invocation results. Store pairs `arg:result` in a dictionary in an attribute of the function object. The function being memoized is:

```
def fibonacci(n):
    assert n >= 0
    if n < 2:
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

Solution

```
def memoize(func):
    func.cache = {}
    def wrapper(n):
        try:
            ans = func.cache[n]
        except KeyError:
            ans = func.cache[n] = func(n)
        return ans
    return wrapper

@memoize
def fibonacci(n):
    """
    >>> print(fibonacci.cache)
    {}
    >>> fibonacci(1)
    1
    >>> fibonacci(2)
    1
    >>> fibonacci(10)
    55
    >>> fibonacci.cache[10]
    55
    >>> fibonacci(40)
    102334155
    """
    assert n >= 0
    if n < 2:
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

Exercise G1 (10 min)

Write a generator function which returns a few values. Launch it. Retrieve a value using `next` (the global function). Retrieve a value using `next` (a method of the generator object). Throw an exception into the generator using `throw` (a method). Look at the traceback.

Exercise G2 (20 min)

You are writing a file browser which displays files line by line. The list of files is specified on the commands line (in `sys.argv`). After displaying one line, the program waits for user input. The user can:

- *press Enter to display the next line*
- *press n + Enter to forget the rest of the current file and start with the next file*
- *or anything else + Enter to display the next line*

The first part is already written: it is a function which displays the lines and queries the user for input. Your job is to write the second part — the generator `read_lines` with the following interface: during construction it is passed a list of files to read. If yields line after line from the first file, then from the second file, and so on. When the last file is exhausted, it stops. The user of the generator can also throw an exception into the generator (`SkipThisFile`) which signals the generator to skip the rest of the current file, and just yield a dummy value to be skipped.

```
class SkipThisFile(Exception):
    "Tells the generator to jump to the next file in list."
    pass

def read_lines(*files):
    """this is the generator to be written

    >>> list(read_lines('exercises.rst'))[:2]
    ['==============================', 'Advanced Python  -- exercises']
    """
    for file in files:
        yield 'dummy line'

def display_files(*files):
    source = read_lines(*files)
    for line in source:
        print(line, end='')
        inp = input()
        if inp == 'n':
            print('NEXT')
            source.throw(SkipThisFile) # return value is ignored
```

Solution

```
def read_lines(*files):
    for file in files:
        for line in open(file):
            try:
```

```
            yield line.rstrip('\n')
        except SkipThisFile:
            yield 'dummy'
            break
```

Exercise M1 (45 min)

The following program writes lines to a file. When ran as a script it tries to add users (`john666`, `john667`, ...) to the user database. It checks that the user name is unique and finds the uid one greater than the previously highest one.

It has been written in a extra-security-concious way which also has the side advantage that it takes quite a lot of time to generate the passphrase, which means that the program is good at exposing race conditions. Unfortunately the author forgot about flushing and locking :(

First check that, indeed, when two instances of the program are run in parallel, the uniqueness constraints are violated.

open three terminals and run in two of them:

```
python add_user.py
```

and in the last one:

```
tail -f /tmp/passwd
```

Pretty soon you should see duplicate entries.

To fix the program

Lock the file with `flock(2)`. What operations have to be protected to ensure correctness? Are the changes to the file flushed to disk before unlocking?

Verify that the uniqueness constraints are now satisfied. (Don't forget to truncate the file before testing the new version of the program.)

If you press ^C at a random moment, will the program always unlock the file?

The program is on the next page and is also available on the wiki page.

```
import crypt, random, itertools

class UserExists(Exception):
    pass

# john:G5yX9p18wFWC.:100:100:/home/john:/bin/sh
PASSWD_FILE = '/tmp/passwd'
def add_user(login, sh='/bin/sh'):
    f = open(PASSWD_FILE, 'a+')
    f.seek(0)

    # find next free uid, verify login is unique
    uid = 0
    for line in f:
        fields_ = line.split(':')
        login_, uid_ = fields_[0], int(fields_[2])
        if uid == uid_:
            uid = uid_ + 1
        if login_ == login:
            raise UserExists(login)

    phrase = ''.join(gen_passwd(10))
    hash = crypt.crypt(phrase[:-2], phrase[-2:])
    print uid, login, phrase

    home = '/home/' + login
    fields = (login, hash, uid, uid, home, sh)
    f.write(u'%s:%s:%d:%d:%s:%s\n' % fields)

def gen_passwd(n):
    symbols = 'ABCDEFGHIJKLMNOPQRSTVUWXYZabcdefghijklmnopqrstvuwxyz0123456789/.'
    for i in xrange(n):
        while True:
            num = random.randrange(0, 1000000)
            if num < 100:
               yield random.choice(symbols)
               break

if __name__ == '__main__':
    for i in itertools.count(666):
        try:
            add_user('john%d'%i)
        except UserExists as e :
            print 'user exists:', e
```

Solution

```
import crypt, random, itertools
import contextlib, fcntl

class UserExists(Exception):
    pass

@contextlib.contextmanager
def flocked(file):
    fd = file.fileno()
    fcntl.lockf(fd, fcntl.LOCK_EX)
    try:
        yield file
    finally:
        fcntl.lockf(fd, fcntl.LOCK_UN)

@contextlib.contextmanager
def flushed(file):
    try:
        yield
    finally:
        file.flush()


# john:G5yX9p18wFWC.:100:100:/home/john:/bin/sh
PASSWD_FILE = '/tmp/passwd'
def add_user(login, sh='/bin/sh'):
    with open(PASSWD_FILE, 'a+') as f:
        with contextlib.nested(flocked(f), flushed(f)):
            f.seek(0)

            # find next free uid, verify login is unique
            uid = 0
            for line in f:
                fields_ = line.split(':')
                login_, uid_ = fields_[0], int(fields_[2])
                if uid == uid_:
                    uid = uid_ + 1
                if login_ == login:
                    raise UserExists(login)

            phrase = ''.join(gen_passwd(10))
            hash = crypt.crypt(phrase[:-2], phrase[-2:])
            print uid, login, phrase

            home = '/home/' + login
            fields = (login, hash, uid, uid, home, sh)
            f.write(u'%s:%s:%d:%d:%s:%s\n' % fields)

def gen_passwd(n):
    symbols = 'ABCDEFGHIJKLMNOPQRSTVUWXYZabcdefghijklmnopqrstvuwxyz0123456789/.'
    for i in xrange(n):
```

```
        while True:
            num = random.randrange(0, 1000000)
            if num < 100:
               yield random.choice(symbols)
               break

if __name__ == '__main__':
    for i in itertools.count(666):
        try:
            add_user('john%d'%i)
        except UserExists as e :
            print 'user exists:', e
```

Exercise D3: plugin registration system (5 min) [optional]

This exercise is to be done at the end if time permits.

This is the plugin registration system from the lecture:

```
class WordProcessor(object):
    def process(self, text):
        for plugin in self.PLUGINS:
            text = plugin().cleanup(text)
        return text

    PLUGINS = []
    ...

@WordProcessor.plugin
class CleanMdashesExtension(object):
    def cleanup(self, text):
        return text.replace('&mdash;', u'\N{em dash}')
```

. . . implement the `plugin` decorator!

Solution

```
class WordProcessor(object):
    ...

    PLUGINS = []

    @classmethod
    def plugin(cls, plugin):
        cls.PLUGINS.append(plugin)
```

Exercise D4 (30 min) [optional]

This exercise is to be done at the end if time permits.

Write a decorator to memoize functions with an arbitrary set of arguments. Memoization is only possible if the arguments are hashable. If the wrapper is called with arguments which are not hashable, then the wrapped function should just be called without caching.

Note: To use `args` and `kwargs` as dictionary keys, they must be hashable, which basically means that they must be immutable. `args` is already a `tuple`, which is fine, but `kwargs` have to be converted. One way is `tuple(sorted(kwargs.items()))`.

```
import functools

def memoize2(func):
    """
    >>> @memoize2
    ... def f(*args, **kwargs):
    ...     ans = len(args) + len(kwargs)
```

```
...     print(args, kwargs, '->', ans)
...     return ans
>>> f(3)
(3,) {} -> 1
1
>>> f(3)
1
>>> f(*[3])
1
>>> f(a=1, b=2)
() {'a': 1, 'b': 2} -> 2
2
>>> f(b=2, a=1)
2
>>> f([1,2,3])
([1, 2, 3],) {} -> 1
1
>>> f([1,2,3])
([1, 2, 3],) {} -> 1
1
"""
func.cache = {}
def wrapper(*args, **kwargs):
    key = (args, tuple(sorted(kwargs.items())))
    try:
        ans = func.cache[key]
    except TypeError:
        # key is unhashable
        return func(*args, **kwargs)
    except KeyError:
        # value is not present in cache
        ans = func.cache[key] = func(*args, **kwargs)
    return ans
return functools.update_wrapper(wrapper, func)
```

Exercise D5 (15 min) [really optional]

Modify `deprecated2` to take an optional argument — a function to call instead of the original function:

```
>>> def eot_new(): return 'EOT NEW'
>>> @deprecated3('using eot_new not {func.__name__}', eot_new)
... def eot(): return 'EOT'
>>> eot()
using eot_new not eot
'EOT NEW'
```

Solution

```
def deprecated3(message, other_func=None):
    """print a message about deprecation once and
    call the original function
```

```
>>> def eot_new(): return 'EOT NEW'
>>> @deprecated3('using eot_new not {func.__name__}', eot_new)
... def eot(): return 'EOT'
>>> eot()
using eot_new not eot
'EOT NEW'
"""
def _deprecated(func):
    count = 0
    repl_func = other_func if other_func else func
    def wrapper(*args, **kwargs):
        nonlocal count
        count += 1
        if count == 1:
            print(message.format(func=func))
        return repl_func(*args, **kwargs)
    return wrapper
return _deprecated
```