Advanced Python — excercises with solutions

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

Excercises use Python 3 syntax! You are encouraged to use `python3.1` to perform the exercises, but `python2.6` can be used just as well, with minor adjustments (mostly `print(...)` -> `print ...`).

Some solutions appear in more than one version: they can be written as classes or as functions. Whatever is easier is better. The behaviour should be identical.

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)
```

Excercise G1 (10 min)

Write a generator function which returns a few values. Launch it. Retrieve a value using `next` (a 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.

Excercise 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  – excercises']
    """
    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. It has been doctored to sleep 1 second in the middle of each line. Unfortunately the author forgot about flushing and locking :(

First write a `fcntl.lockf` context manager.

Solution:

```
class flocked:
    def __init__(self, fd):
        self.fd = fd
    def __enter__(self):
        fcntl.lockf(self.fd, fcntl.LOCK_EX)
    def __exit(self, *args):
        fcntl.lockf(self.fd, fcntl.LOCK_UN)
```

Convert `add_user` below to use your new context manager stacked with `flushed` to fix the program.

To check: open two terminals and run (note the ampersand to run in parallel):

```
python3.1 add_user.py /tmp/passwd 100&
python3.1 add_user.py /tmp/passwd 100
```

in one of them and in the other one

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

The program (also available on the wiki page):

```
import sys, io, fcntl, time, datetime

def add_user(login, uid, gid, name, home, shell='/bin/bash'):
    fields = login, 'x', str(uid), str(gid), name, home, shell
    f = open(PASSWD_FILE, 'a')

    f.write(':'.join(fields[:3]) + ':')
    f.flush()
    time.sleep(1)
    f.write(':'.join(fields[3:]) + '\n')

def main(prog_name, file_name, count):
    global PASSWD_FILE
    PASSWD_FILE = file_name
    for i in range(int(count)):
         now = datetime.datetime.now()
         uid = now.minute * 60 + now.second
         gid = now.microsecond//1000
         add_user('login', uid, gid, 'name', 'home')
         print('added uid={}'.format(uid))

if __name__ ==  '__main__':
    sys.exit(main(*sys.argv))
```

Exercise D3: plugin registration system (15 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:
    """
    PLUGINS = []
    def process(self, text):
        for plug in self.PLUGINS:
            text = plug().cleanup(text)
        return text

@register(WordProcessor.PLUGINS)
class CleanMdashesExtension():
    def cleanup(self, text):
        return text.replace('&mdash;', '\u2014')
```

…implement the `register` decorator

Solution

```
>>> def register(where):
...     def helper(cls):
...         where.append(cls)
...         return cls
...     return helper
>>> class WordProcessor:
...       PLUGINS = []
...       def process(self, text):
...           for plug in self.PLUGINS:
...               text = plug().cleanup(text)
...           return text
...
>>> @register(WordProcessor.PLUGINS)
... class CleanMdashesExtension():
...       def cleanup(self, text):
...           return text.replace('&mdash;', '\u2014')
...
>>> WordProcessor().process('hello &mdash; hello')
'hello — hello'
```

Excercise 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()))`.

Solution

```
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
```