Python II

Gopas

Review

- List, Dictionaries, Tuples, Sets
- Mutable vs. Imutable types
- Conditions, Loops
- Functions, lambda expressions
- List comprehension
- Exceptions

OOP

- Classes: User defined types of objects (including their methods, attributes, relations to other objects). Can be instantiated into an object / is a ‘blueprint’ that describes how to build an object.

- Python does not enforce OOP (unlike Java), but we need to understand at least what is going on.

- Class definitions contain methods (which are functions defined in the class scope), class attributes, and a docstring.

OOP

- Class
- Instance
- Instance variable, method
- Attribute
- Inheritance
- Encapsulation
- Polymorphism

OOP

```
class MyFirst:
        ‘‘‘Doc string‘‘‘
        body

instance=MyFirst()
```

OOP

- Python supports large amount of special methods

- creation/destroy of objects
- aritmetic operations
- logic operations (comparisions)
- work with sequences
- work with attributes

OOP

```
class Person:
    def __init__(self,name,age):
        self.name=name
        self.age=age
    def __str__(self):
        return(self.name)
    def __gt__(self,other):
        if (self.age>other.age):
            return True
        return False
    def __add__(self,other):
        return self.age+other.age
    def printall(self):
        print (“Name : %s, age : %d" % (self.name,self.age))

bob=Person(“Bob",20)
alice=Person(“Alice",19)
print(bob+alice)
```

Class variables

```
class Person:
    Person_id=1

    def __init__(self,name,age):
        self.name=name
        self.age=age
        self.cid=Person.Person_id
        Person.Person_id+=1

    def printall(self):
        print ("Name : %s, age : %d, id : %d" % (self.name,self.age,self.cid))

bob=Person(“Bob",20)
alice=Person(“Alice",19)

bob.printall()
alice.printall()
```

Class methods

```
class Person:
    Person_id=1
    def __init__(self,name,age):
        self.name=name
        self.age=age
        self.cid=Person.Person_id
        Person.Person_id+=1
    def resetPerson(cls):
        cls.Person_id=1
    resetPerson=classmethod(resetPerson)
    def printall(self):
        print ("Name : %s, age : %d, id : %d" %
(self.name,self.age,self.cid))
```

```
bob=Person(“Bob",20)
bob.resetPerson()
alice=Person(“Alice",19)

bob.printall()
alice.printall()
```

Class method as decorator

```
class Person:
    Person_id=1

    def __init__(self,name,age):
        self.name=name
        self.age=age
        self.cid=Person.Person_id
        Person.Person_id+=1

    @classmethod
    def resetPerson(cls):
        cls.Person_id=1

    def printall(self):
        print ("Name : %s, age : %d, id : %d" % (self.name,self.age,self.cid))
```

Decorators

- A decorator is the name used for a software design pattern. Decorators dynamically alter the functionality of a function, method, or class without having to directly use subclasses or change the source code of the function being decorated.
- @classmethod
- def foo (arg1, arg2):
- ….
- @property

Functors

- Functor is simply a mapping from one type of data to another.
- In Python a *function object* is an object reference to any callable, such as a function, a lambda function, or a method. The definition also includes classes, since an object reference to a class is a callable that, when called, returns an object of the given class—for example, x = int(5). In computer science a *functor* is an object that can be called as though it were a function, so in Python terms a functor is just another kind of function object. Any class that has a __call__() special method is a functor.

Getters, setters, property

```
class Person (object):
  def __init__(self,name,age):
    self.__name=name
    self.__age=age
  def printit(self):
    print (“Name is : %s, age is : %d" % (self.__name,self.__age))
  @property
  def name(self):
    return self.__name
  @name.setter
  def name(self,name):
    self.__name=name

bob=Person(“Bob",20)
print(bob.name)
bob.name=“BOB"
```

Metaclasses

- A metaclass is to a class what a class is to an instance; that is, a metaclass is used to create classes, just as classes are used to create instances. And just as we can ask whether an instance belongs to a class by using isinstance(), we can ask whether a class object (such as dict, int, or SortedList) inherits another class using issubclass().
- One use of metaclasses is to provide both a promise and a guarantee about a class’s API. Another use is to modify a class in some way (like a class decorator does). And of course, metaclasses can be used for both purposes at the same time.

Iterable/Iterator

- Iterable is everything what can be used to iterate over:
  - for var in *iterable*:
  - for i in ‘cau’: print i
- Iterator is object which remembers state where is during and between iteration calls
- s=“Bye”
- i=iter(s)
- next(i) #‘B’
- next(i) #‘y’
- next(i) #’e’
- next(i) #exception StopIteration

Iterator

```
class firstn(object):
    def __init__(self, n):
        self.n = n
        self.num = 0
    def __iter__(self):
        return self
    # Python 3 compatibility
    def __next__(self):
        return self.next()
    def next(self):
        if self.num < self.n:
            cur, self.num = self.num, self.num+1
            return cur
        else:
            raise StopIteration()


sum_of_first_n = sum(firstn(100))
```

Generator

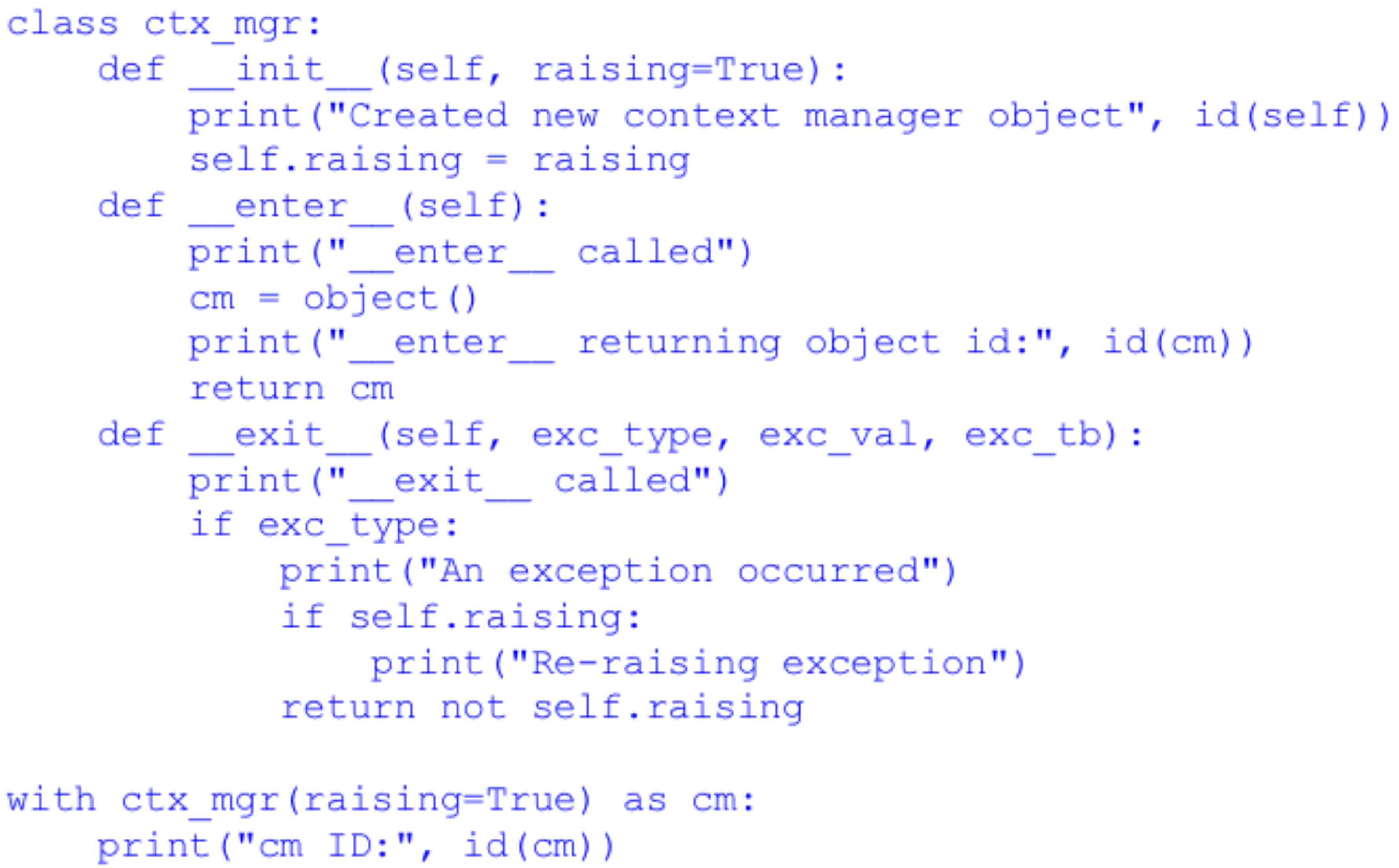
```
def firstn(n):
    num = 0
    while num < n:
        yield num
        num += 1

  sum_of_first_n = sum(firstn(100))
```

Context Managers

- with **object1** [**as** name1][, **object2** [**as** name2]] ...:
- [indented suite]

- The Context Manager Protocol: __enter__() and __exit__()
- The **with** statement has rules for interacting with the object it is given as a context manager. It processes **with expr** by evaluating the expression and saving the resulting *context manager object*. The context manager's __enter__() method is then called, and if the **as name** clause is included, the result of the method call is bound to the given name. Without the **as name** clause, the result of the __enter__() method is not available. The indented suite is then executed.

Context Managers



```
class ctx_mgr:
    def __init__(self, raising=True):
        print("Created new context manager object", id(self))
        self.raising = raising
    def __enter__(self):
        print("__enter__ called")
        cm = object()
        print("__enter__ returning object id:", id(cm))
        return cm
    def __exit__(self, exc_type, exc_val, exc_tb):
        print("__exit__ called")
        if exc_type:
            print("An exception occurred")
            if self.raising:
                print("Re-raising exception")
            return not self.raising

with ctx_mgr(raising=True) as cm:
    print("cm ID:", id(cm))
```

Coroutines

- Coroutines are functions whose processing can be suspended and resumed at specific points. So, typically, a coroutine will execute up to a certain statement, then suspend execution while waiting for some data. At this point other parts of the program can continue to execute (usually other coroutines that aren’t suspended).

Coroutines

```
@coroutine
def regex_matcher(receiver, regex):
  while True:
    text = (yield)
    for match in regex.finditer(text):
      receiver.send(match)
```

Work with databases

Python defines Python Database API Specification v2.0

Relational databases are the most widely used type of database, storing information as tables containing a number of rows.

Example SQlite

Work with databases

```
import sqlite3

conn=sqlite3.connect(“phones.sqlite")
cursor=conn.cursor()
cursor.execute("select * from phones")

for record in cursor.fetchall():
    print("Name : %s, phone number : %s" %(record[0],record[1]))

conn.close()
```

Work with databases

```
import sqlite3

conn=sqlite3.connect(“phones.sqlite")
cursor1=conn.cursor()
cursor2=conn.cursor()

cursor1.execute("insert into phones values (‘Police‘,‘911')")
conn.commit()
cursor2.execute("select * from phones")

for record in cursor2.fetchall():
    print("Name : %s, cislo : %s" %(record[0],record[1]))

conn.close()
```

Regular expressions

- Complex searching and substitutions
- Regular expression is not specially quoted in Python
  - Be careful on \
  - Use raw string r”\.html$”
- Anchors ^,$
- Quantifiers *+? {}
- Character sets [] [^], interval a-z
- \d \w \z
- Grouping () \1..\99

import re

- Compilation re.compile(re,[modifiers])
- Methods of object representing RE
  - match
  - search
  - findall
  - finditer
- Or you can use match(re,string), search(re,string)…

Match object

- Methods
    - start()
    - end()
    - group()
    - span()
- Named group (?P<name>…)

- m=re.compile("a+")
- s="accaabaaavvv”
- print m.findall(s) #['a', 'aa', 'aaa']

Substitution with RE

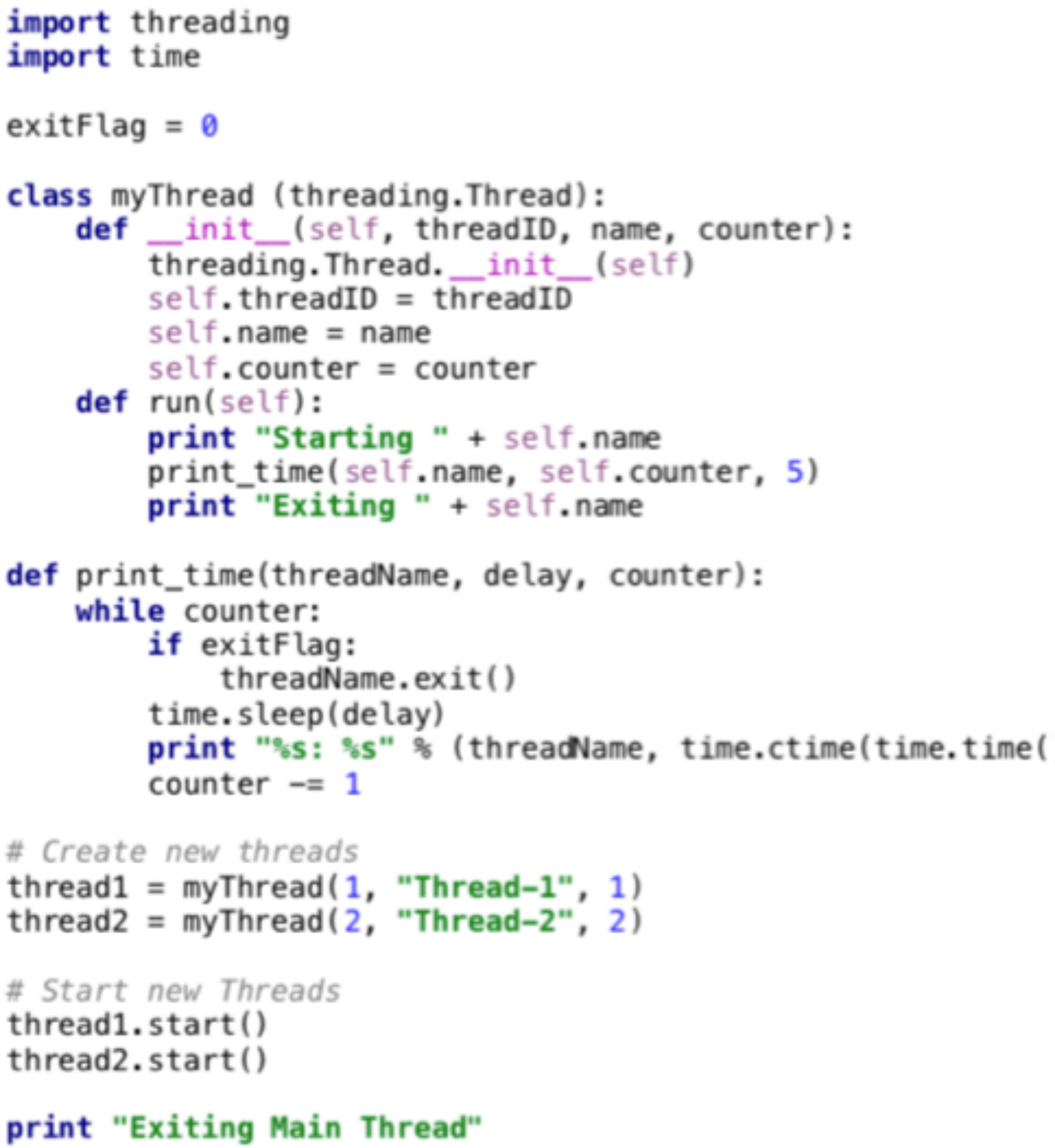
- Methods of object representing RE
  - split(string[, maxsplit=0])
  - sub(replacement, string[, count=0])
  - subn(replacement, string[, count=0])

- m=re.compile("a+")
- s="accaabaaavvv”
- print m.sub('A',s) #AccAbAvvv

Parallel programming

- **import thread**
- **import time**
- ***# Define a function for the thread***
- **def print_time(threadName, delay):**
- **count = 0**
- **while count < 5:**
  time.sleep(delay)
  count += 1
  print "%s: %s" % (threadName, time.ctime(time.time()))
- ***# Create two threads as follows***
- **try:**
  thread.start_new_thread(print_time, ("Thread-1", 2,))
- **thread.start_new_thread(print_time, ("Thread-2", 4,))**
- **except:**
- **print "Error: unable to start thread“**
- **while 1:**

Parallel programming



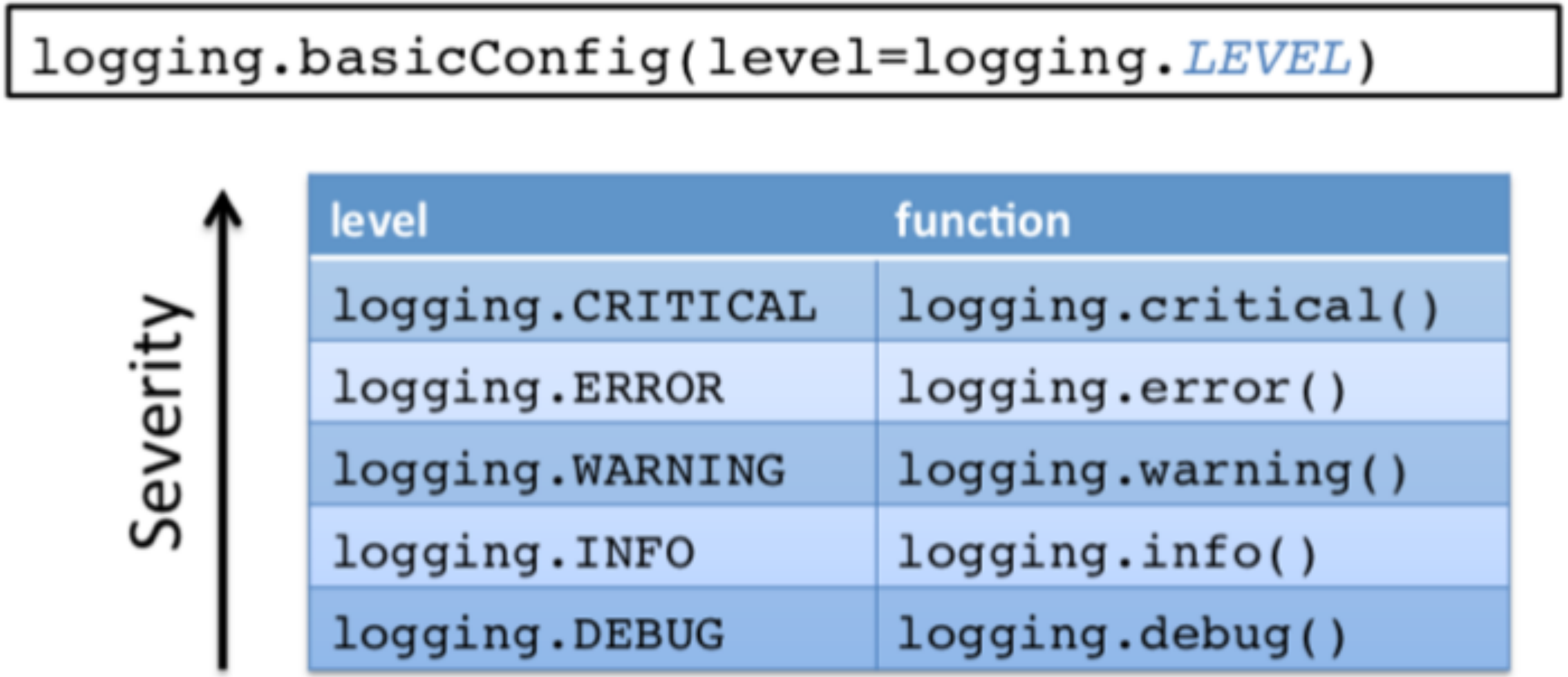
```
import threading
import time

exitFlag = 0

class myThread (threading.Thread):
    def __init__(self, threadID, name, counter):
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.name = name
        self.counter = counter
    def run(self):
        print "Starting " + self.name
        print_time(self.name, self.counter, 5)
        print "Exiting " + self.name

def print_time(threadName, delay, counter):
    while counter:
        if exitFlag:
            threadName.exit()
        time.sleep(delay)
        print "%s: %s" % (threadName, time.ctime(time.time(
        counter -= 1

# Create new threads
thread1 = myThread(1, "Thread-1", 1)
thread2 = myThread(2, "Thread-2", 2)

# Start new Threads
thread1.start()
thread2.start()

print "Exiting Main Thread"
```

logging module

- Logging module to log errors and debugging messages
- Provides central control over debugging output

```
import logging
logging.basicConfig(level = logging.DEBUG)
def mirror(lst):
  ret = []
  for i in range(len(lst)):
    ret.append(lst[-i - 1])
    logging.debug(“list for i={0}: {1} ".format(i, lst[-i - 1]))
  return lst + ret
```

logging



```
logging.basicConfig(level=logging.LEVEL)
```

level	function
logging.CRITICAL	logging.critical()
logging.ERROR	logging.error()
logging.WARNING	logging.warning()
logging.INFO	logging.info()
logging.DEBUG	logging.debug()

logging

- Can output messages to a log file
- logging.basicConfig(level=logging.DEBUG, filename = ‘bugs.log’)

- Can add time
- logging.basicConfig(level=logging.DEBUG, filename=‘bugs.log’, format=‘%(asctime)s %(message)’)

Functional style programming

- Functional programming are these concepts: *mapping*, *filtering*, and *reducing*

- list(map(lambda x: x ** 2, [1, 2, 3, 4]))
- [x ** 2 for x in [1, 2, 3, 4]]

- list(filter(lambda x: x > 0, [1, -2, 3, -4]))
- [x for x in [1, -2, 3, -4] if x > 0]

Functional style programming

- functools.reduce(lambda x, y: x * y, [1, 2, 3, 4])
- functools.reduce(operator.mul, [1, 2, 3, 4])

- functools.reduce(operator.add, (os.path.getsize(x) for x in files))
- functools.reduce(operator.add, map(os.path.getsize, files))

Functional style programming

- functools.reduce(operator.add, map(os.path.getsize, filter(lambda x: x.endswith(".py"), files)))
- functools.reduce(operator.add, map(os.path.getsize, (x for x in files if x.endswith(".py"))))
- functools.reduce(operator.add, (os.path.getsize(x) for x in files if x.endswith(".py")))

Functional style programming

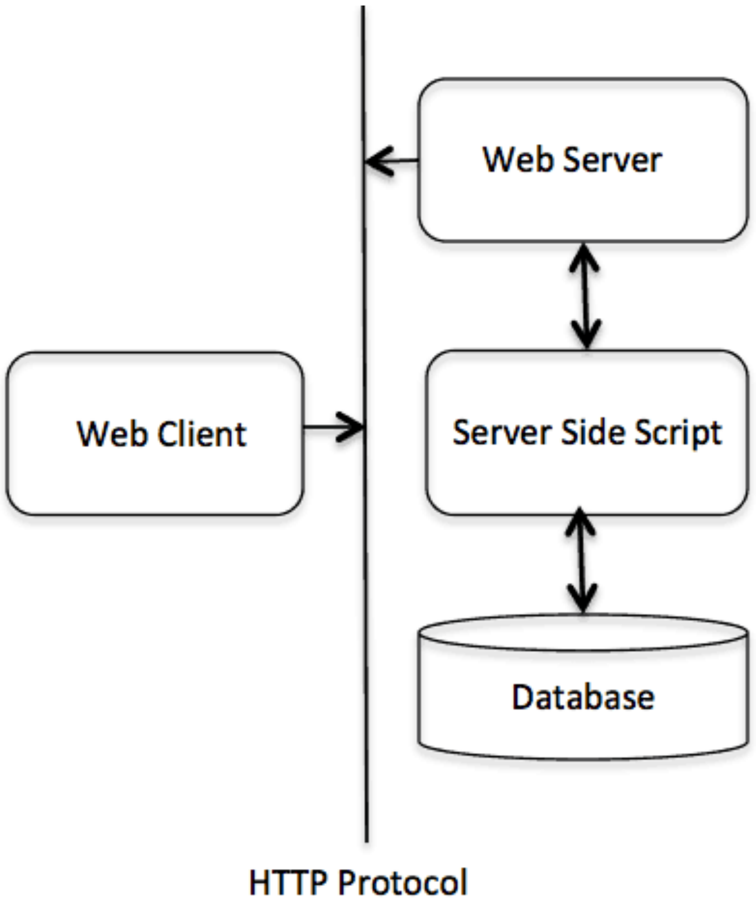
- for value in itertools.chain(data_list1, data_list2, data_list3):
-   total += value

Anonymous functions

- The result of a lambda expression is ananonymous function. When a lambda function is called it returns the result of computing the *expression* as its result.

- s = lambda x: "" if x == 1 else "s"

Python CGI


Web Server
Web Client
Server Side Script
Database
HTTP Protocol

Python CGI

- Traditional CGI scripts in Unix shell, Perl, awk, C/C++
- Python standard modules cgi and cgitb
- Apache http server /var/www/cgi-bin
- os.environ
- form = cgi.FieldStorage()
- value = form.getvalue(“param_name“)

Flask

- Web microframework
- routing, debugging, and Web Server Gateway Interface (WSGI) subsystems
- template support is provided by Jinja2
- User authentication, form validations are available throught extensions
- Instalation: pip install flask

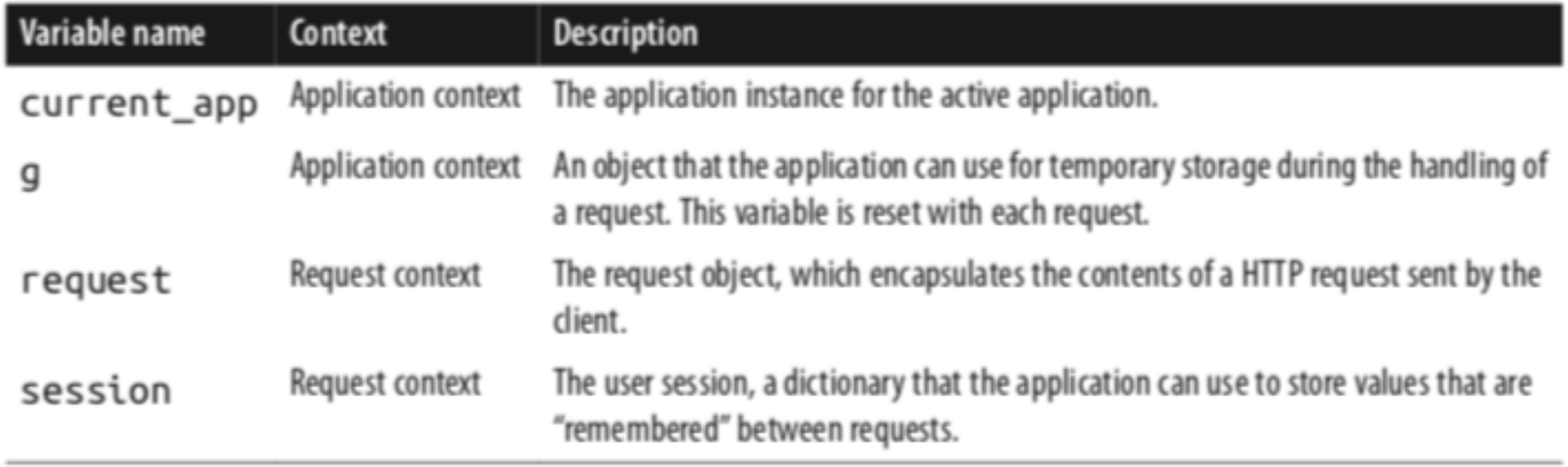
Flask sample

```
from flask import Flask
app = Flask(__name__)

@app.route('/‘)
def index():
  return '<h1>Hello World!</h1>‘

if __name__ ==  “__main__“:
  app.run(debug=True)  #host=‚0.0.0.0‘, port=8080
```

URL parameters

```
from flask import request
@app.route('/')
def index():
  user_agent = request.headers.get('User-Agent‘)
  return '<p>Your browser is %s</p>' % user_agent
```

Application and request contexts



Variable name	Context	Description
current_app	Application context	The application instance for the active application.
g	Application context	An object that the application can use for temporary storage during the handling of a request. This variable is reset with each request.
request	Request context	The request object, which encapsulates the contents of a HTTP request sent by the client.
session	Request context	The user session, a dictionary that the application can use to store values that are “remembered” between requests.

View function error status

```
@app.route('/‘)
def index():
  return '<h1>Bad Request</h1>', 400
```

Cookies

- Use response object

```
from flask import make_response
@app.route('/‘)
def index():
  response = make_response('<h1>This document carries a cookie!</h1>‘)
  response.set_cookie('answer', '42‘)
  return response
```

The Jinja2 Template Engine

- templates/user.html
- <h1>Hello, {{ name }}!</h1>

- Rendering

```
from flask import Flask, render_template
@app.route('/index‘)
def index():
  return render_template('index.html')
@app.route('/user/<name>‘)
def user(name):
  return render_template('user.html', name=name)
```

Variables

<p>A value from a dictionary: {{ mydict['key'] }}.**</p>**
<p>A value from a list: {{ mylist[3] }}.**</p>**
<p>A value from a list, with a variable index: {{ mylist[myintvar] }}.**</p>**

<p>A value from an object's method: {{ myobj.somemethod() }}.**</p>**

Filters (capitalize,lower,upper,trim,title,striptags,safe)

Hello, {{ name|capitalize }}

Jinja2 control structures

```
{% if user %}
  Hello, {{ user }}
{% else %}
  Hello, Stranger
{% endif %}
```

```
<ul>
  {% for comment in comments %}
      <li>{{ comment }}</li>
  {% endfor %}
</ul>
```

Error handling

```
@app.errorhandler(404)
def page_not_found(e):
  return render_template('404.html'), 404
@app.errorhandler(500)
def internal_server_error(e):
  return render_template('500.html'), 500
```