Python and Design Patterns

丁来强 (Lai Qiang Ding)

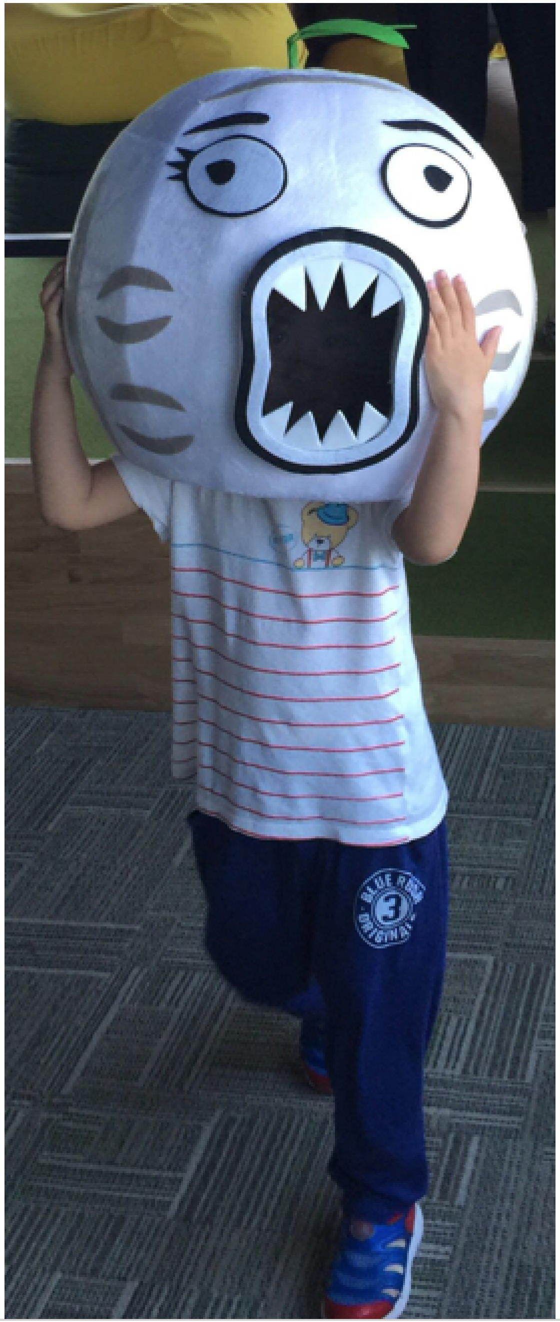
wjo1212

wjo1212@163.com

LaiQiangDing

About Me

Father of a 4 years' boy



About Me

- Worked for 10+ years.
- @Splunk, Tech Lead (Sr. Technical Manager)

Agenda

- Misunderstanding of Design Patterns
- General Idea of Python and Design Patterns
- **Creational** Design Patterns Practice
  - 4 GOF and 2 other patterns
- **Structural** Design Patterns Practice
  - 3 GOF, 2 FP and 1 other patterns
- **Behavioral** Design Patterns Practice
  - 5 GOF, 3 FP and 2 other patterns
- **Concurrent** Patterns

    - 6 current patterns

Patterns Covered (30+)

- GOF Patterns (13)
    - Abstract Factory, Factory Method
    - Singleton, Prototype, Builder
    - Decorator,
    - Proxy, Bridge
    - Template Method
    - Strategy
    - Observer
    - Iterator, Visitor

Patterns Covered (30+)

- FP Patterns (5)
    - Currying
    - Compose
    - High Order Function
    - Memorization
    - Monads

Patterns Covered (30+)

- Concurrency Patterns (6)
    - Reader Writer Lock
    - Guarded suspension
    - Advanced Producer and Consumer
    - Promise (Future)
    - Thread Pool
    - Executor Service

Patterns Covered (30+)

- Other Popular Patterns (6)
  - RAII Idiom
  - Dependency Injection
  - MonoState
  - Mixin
  - Double Dispatch
  - Mock-up

You will learn

- Understand design pattern correctly, know when, how to use it
- Know most popular design pattern practices used in Python from GOF, Concurrency, Functional and Other popular ones.
- Know how to further learn design patterns and architectual pattern systematically

Note

Some patterns and technology used are also comprehensively and deeply discussed in my previous talks:

To get a comprehensive understanding of **Functional Programming** with Python and relative patterns in detail,

- Refer to my talk " Effecitve Python Functional Programming" (https://github.com/wjo1212/ChinaPyCon2015) in PyCon 2015 Beijing

To get a comprehensive understanding of **Decorator, Metaclass, Magic Methods and other Hooking technology** with Python,

- Refer to my talk Python Hooking, Patching and Injection (https://github.com/wjo1212/ChinaPyCon2016) in PyCon 2016 Shenzhen

1. Misunderstanding of Design Patterns

1.1. General Principle of Software Development

- YAGNI: "You aren't gonna need it"
  - https://en.wikipedia.org/wiki/You_aren%27t_gonna_need_it (https://en.wikipedia.org/wiki/You_aren%27t_gonna_need_it)
- KISS: “Keep it simple stupid"
  - https://en.wikipedia.org/wiki/KISS_principle (https://en.wikipedia.org/wiki/KISS_principle)

1.2. What is Design Patterns

- not data structures, nor algorithms
- not domain-specific architectures for entire subsystems
- just: "descriptions of communicating objects and classes that are customized to **solve a general design problem in a particular context**" [Gof4]

1.3. Misunderstanding

- “design patterns should be used right from the start when writing code.”
  - “It is not unusual to see developers struggling with which pattern they should use in their code, even if they haven't first tried to solve the problem in their own way”
- “design patterns should be used everywhere. "
  - This results in creating complex solutions with unnecessary interfaces and hierarchies.

1.4. More practical points

- “The most important part of a design pattern is probably **its name**. The benefit of naming all patterns is that we have, on our hands, a common vocabulary to **communicate** [GOF95, page 13]. ”
- “Design patterns are discovered (in contrast to invented) as better solutions over **existing solutions. If you have no existing solution, it doesn't make sense to look for a better one.** ”

- “Do no treat design patterns as a panacea because they are not. They must be used **only if there is proof that your existing code "smells", and is hard to extend and maintain.** ”

- some more aggresive points:
  - many "classic" DP (for C++ or Java) are "**workarounds** against static typing" (cfr: Alpert, Brown, Woolf, "The DPs Smalltalk Companion", Addison-Wesley DP Series)
  - Features in language reduce/remove patterns, and thus shorten code.
  - There are still patterns, and where those patterns exist, that's a ripe place for a new language feature

1.5. Idioms

- An idiom is an idea to work around the quirks of a language.
  - Double Checked Locking for Singleton in multi-thread context in Java/C++
  - Smart Pointer in C++
  - Pimpl Idiom in C++
  - Final Idiom in C++

- Some other points: Programming Idiom as a low-level Design Pattern.
- When a language directly supports the patterns, it will become an idiom for that languages.
  - Resource Acquisition Is Initialization (RAII) support by Python (via with/context manager) or Java7 (via try-with-resource)
  - for and Iterator in Python/C++11/Java5
  - Curountine support in Python
  - Coroutine support in Go
  - Observer support in C#

1.5.1. Resource Acquisition Is Initialization (RAII) idiom

Intent

Resource Acquisition Is Initialization pattern can be used to implement exception safe resource management.

Applicability

Use the Resource Acquisition Is Initialization pattern when

Implementation

In [44]:
```
class DBCon(object):
    def __init__(self, con_str):
        self.con_str = con_str

    def __enter__(self,):
        print "connect to remote db connection..."

    def __exit__(self, exc, val, trace):
        print "close the remote connection...."
```

In [45]:
```
with DBCon("ip=...&port=..."):
    1/0
```

```
connect to remote db connection...
close the remote connection....

---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-45-bb27d3223b3c> in <module>()
      1 with DBCon("ip=...&port=..."):
----> 2     1/0

ZeroDivisionError: integer division or modulo by zero
```

2. Python and Design Pattern

2.1. Python is:

- Dynamic Language
  - Support Duck-typing
- OOP Language
  - Support OO features completely
- FP Friendly
  - Functions are first-class citizens
  - Support most FP features
- Has lots of brilliant built-in features
  - Some are directly for some patterns
  - e.g. iterator, decorator, prototype and FP features

2.2. Misunderstanding

- Design Patterns are for **OO**, Python is dynamic language which doesn't need it
- Design Patterns are **independent with languages**

2.3. More Practical Points:

2.3.1. Design Pattern is NOT indepent with language

- "Point of view affects one's interpretation of what is and isn't a pattern... **choice of programming language is important** because it **influences** one's point of view" [Gamma et al, "Design Patterns"]
- In the concept of original "Design Pattern", when building with wood, concrete, many patterns remains w/small changes, but **many appear, disappear, or change deeply**.

2.3.2 Fist-class types and First-class functions make many GOF patterns invisible or much simpler

First-Class types:

- class/types can be used and operated on where any other value or object can be used
- Types or Classes are objects at run-time

- A variable can have a type as a value
- A type or class can be created/modified at run-time
- There are functions to manipulate types/classes (and expressions to create types without names)
- No need to build extra dynamic objects just to hold types, because the type objects themselves will do

Detail:

- Type/class is intrinsically factory
  - __new__, can be injected directly w/boilerplate factory way
  - Metaclass __call__, __new__, __init__ provides two phase construction
  - There's no new keyword, calling way is same as function (transparent)

Examples:

- Abstract-Factory, Flyweight, Factory-Method, State, Proxy, Bridge, Chain-Of-Responsibility

First-Class functions:

- Functions are objects too
- Functions are composed of methods
- There are operations on functions (compose, conjoin)
- Code is organized around functions as well as classes
- Function closures capture local state variables (Objects are state data with attached behavior; Closures are behaviors with attached state data and without the overhead of classes.)

Examples:

- Command, Strategy, Template-Method, Visitor, Builder

2.4. An example (Strategy Pattern)

2.4.1. Classic implementation

In [104]:
```
import json
import re
from abc import ABCMeta, abstractmethod

class IDataParser(object):
    __metaclass__ = ABCMeta

    @abstractmethod
    def parse(cls, content): pass

class IDataLoader(object):
    __metaclass__ = ABCMeta

    @abstractmethod
    def get_content(self): pass

class ConfigData(object):
    def __init__(self, loader, parser):
        assert isinstance(loader, IDataLoader) and isinstance(parser, IDataParser)
        self.data = parser.parse(loader.get_content())

    def __getitem__(self, item):
        return self.data.get(item, None)
```

In [105]:
```
class JsonParser(IDataParser):
    def parse(cls, content):
        return json.loads(content)

class KvParser(IDataParser):
    def parse(cls, content):
        return dict(re.findall(r'(\w+)\s*=\s*"?([\w\s]+)"?', content))

class S3Storage(IDataLoader):
    def get_content(self):
        # read content from AWS S3
        return '{ "d1": "s3 d1", "d2": "s3 d2" }'

class FileStorage(IDataLoader):
    def get_content(self):
        # read content from file path
        return 'd1="file d1", d2="file d2"'
```

In [106]:
```
s3_config = ConfigData(S3Storage(), JsonParser())
print s3_config['d1']
print s3_config['d2']

file_config = ConfigData(FileStorage(), KvParser())
print file_config['d1']
print file_config['d2']
```

```
s3 d1
s3 d2
file d1
file d2
```

2.4.2. implementation with duck typing

In [1]:
```
class ConfigData(object):
    def __init__(self, loader, parser):
        self.data = parser.parse(loader.get_content())

    def __getitem__(self, item):
        return self.data.get(item, None)

class JsonParser(object):
    def parse(cls, content):
        return json.loads(content)

class KvParser(object):
    def parse(cls, content):
        return dict(re.findall(r'(\w+)\s*=\s*"?([\w\s]+)"?', content))

class S3Storage(object):
    def get_content(self):
        return '{ "d1": "s3 d1", "d2": "s3 d2" }'

class FileStorage(object):
    def get_content(self):
        return 'd1="file d1", d2="file d2"'
```

In [109]:
```
s3_config = ConfigData(S3Storage(), JsonParser())
print s3_config['d1']
print s3_config['d2']

file_config = ConfigData(FileStorage(), KvParser())
print file_config['d1']
print file_config['d2']
```

```
s3 d1
s3 d2
file d1
file d2
```

2.4.3. Simpler way using functions

In [110]:
```
import json
import re

def parse_json(content):
    return json.loads(content)

def parse_kv(content):
    return dict(re.findall(r'(\w+)\s*=\s*"?([\w\s]+)"?', content))

def load_s3():
        # read content from AWS S3
        return '{ "d1": "s3 d1", "d2": "s3 d2" }'

def load_file():
        # read content from file path
        return 'd1="file d1", d2="file d2"'
```

In [111]:
```
s3_config = parse_json(load_s3())
print s3_config['d1']
print s3_config['d2']

file_config = parse_kv(load_file())
print file_config['d1']
print file_config['d2']
```

```
s3 d1
s3 d2
file d1
file d2
```

3. Creational Design Patterns Practice

- Concern the ways and means of object instantiation

Pattern Coverd:

- **GOF Patterns:**
  - Abstract Factory

    - Factory Method
    - Singleton
    - Prototype
- **Other related:**
    - **Dependency Injection**
    - **MonoState**

3.1. Abstract Factory

Also known as

Kit

Intent

Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Applicability

Use the Abstract Factory pattern when

- a system should be configured with one of multiple families of products
- you want to provide a class library of products, and you want to reveal just their interfaces, not their implementations

Implementation

Just using **different modules** is fine in python

just like the "os" module providing same functions for windows, linux and mac etc.

In [86]:
```python
import code.mac_theme as util

util.create_window('hello world')
util.start_process('calendar')
```

```
create window on mac:  hello world
start process on mac:  calendar
```

In [87]:
```python
import code.windows_theme as util

util.create_window('hello world')
util.start_process('notepad.exe')
```

```
create window on windows:  hello world
start process on windows:  notepad.exe
```

3.2. Factory Method

Also known as

Virtual Constructor

Intent

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Applicability

Use the Factory Method pattern when

- a class can't anticipate the class of objects it must create
- a class wants its subclasses to specify the objects it creates
- classes delegate responsibility to one of several helper subclasses

Implementation

In [ ]:
```
import json
class Student(object):
    def __init__(self, name=None, sex=None, age=None):
        self.name = name
        self.sex = sex
        self.age = age

class Teacher(object):
    def __init__(self, name=None, sex=None, age=None, role=None):
        self.name = name
        self.sex = sex
        self.age = age
        self.role = role
```

In [157]:
```
cls_map = {
    'student': (Student, 'code/student.json'),
    'teacher': (Teacher, 'code/teacher.json')
}

def people_factory(type):
    assert type in cls_map
    cls, data_path = cls_map[type]
    with open(data_path) as f:
        data = json.load(f)
        return cls(**data)
```

In [159]:
```
s = people_factory('student')
print s.name, s.age, s.sex

t = people_factory('teacher')
print t.name, t.age, t.sex, t.role
```

```
Xiao Ming 10 True
Mr. Wang 10 True English
```

3.3. Dependency Injection (Other)

Intent

Dependency Injection is used for one or more dependencies (or services) are injected, or passed by reference, into a dependent object (or client) and are made part of the client's state. The pattern separates the creation of a client's dependencies from its own behavior, which allows program designs to be loosely coupled and to follow the inversion of control and single responsibility principles.

Applicability

Use the Dependency Injection pattern when

- when you need to remove knowledge of concrete implementation from object to enable unit testing of classes in isolation using mock objects or stubs

3.3.1. Implementation (Class level)

In [14]:
```
import json
import re

class ConfigData(object):
    def __init__(self, loader, parser):
        self.data = parser.parse(loader.get_content())
    def __getitem__(self, item):
        return self.data.get(item, None)

class JsonParser(object):
    def parse(self, content):
        return json.loads(content)

class KvParser(object):
    def parse(self, content):
        return dict(re.findall(r'(\w+)\s*=\s*"?([\w\s]+)"?', content))

class S3Storage(object):
    def get_content(self):
        return '{ "d1": "s3 d1", "d2": "s3 d2" }'

class FileStorage(object):
    def get_content(self):
        return 'd1="file d1", d2="file d2"'
```

Traddtional Usage

In [65]:
```
config = ConfigData(S3Storage(), JsonParser())

print config['d1']
print config['d2']
```

```
s3 d1
s3 d2
```

Implicit Depedency Injection

In [15]:

```
import pinject
import __main__

class Loader(S3Storage): pass
class Parser(JsonParser): pass

di = pinject.new_object_graph(modules=[__main__])

config = di.provide(ConfigData)

print config['d1']
print config['d2']
```

```
s3 d1
s3 d2
```

Explicit Dependency Injection

In [ ]:

```
class ConfigBindingSpec(pinject.BindingSpec):
    def __init__(self, type):
        self.type = type

    def configure(self, bind):
        if self.type == "s3":
            bind('loader', to_class=S3Storage)
            bind('parser', to_class=JsonParser)
        elif self.type == "file":
            bind('loader', to_class=FileStorage)
            bind('parser', to_class=KvParser)
```

In [67]:

```
di = pinject.new_object_graph(modules=[__main__], binding_specs=[ConfigBindingSpec('s3')])

config = di.provide(ConfigData)

print config['d1']
print config['d2']
```

```
s3 d1
s3 d2
```

3.3.2. Implementation (Functional Level)

In [21]:

```
import json
import re
import pinject
import __main__

class ConfigData(object):
    def __init__(self, load_fn, parse_fn):
        self.data = parse_fn(load_fn())
    def __getitem__(self, item):
        return self.data.get(item, None)

def parse_json(content):
    return json.loads(content)

def parse_kv(content):
    return dict(re.findall(r'(\w+)\s*=\s*"?([\w\s]+)"?', content))

def load_s3():
        return '{ "d1": "s3 d1", "d2": "s3 d2" }'

def load_file():
        return 'd1="file d1", d2="file d2"'
```

Traddtional Usage

In [71]:

```
s3_config = ConfigData(load_s3, parse_json)
print s3_config['d1']

file_config = ConfigData(load_file, parse_kv)
print file_config['d1']
```

```
s3 d1
file d1
```

Dependency Injection

In [19]:
```
class ConfigBindingSpec(pinject.BindingSpec):
    def __init__(self, type):
        self.type = type

    def configure(self, bind):
        if self.type == "s3":
            bind('load_fn', to_instance=load_s3)
            bind('parse_fn', to_instance=parse_json)
        elif self.type == "file":
            bind('load_fn', to_class=load_file)
            bind('parse_fn', to_class=parse_kv)
```

In [72]:
```
di = pinject.new_object_graph(modules=[__main__], binding_specs=[ConfigBindingSpec('s3')])

config = di.provide(ConfigData)

print config['d1']
print config['d2']
```

```
s3 d1
s3 d2
```

3.4. Singleton

Intent

Ensure a class only has one instance, and provide a global point of access to it.

Applicability

Use the Singleton pattern when

- there *must* be exactly one instance of a class, and it must be accessible to clients from a well-known access point
- when the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code

Note

Singleton have been widly abused and become an **Anti-Pattern**

Implementation

Directly puting the singleton into a module would be OK.

Note: Importing module itself is thread-safe

```
def import_module(mod_name):
    if mod_name in sys.modules:
        return sys.modules[mod_name]

    for dir_name in sys.path:
        print dir_name
        file_name = op.join(dir_name, mod_name + ".py")

        m = _exec_file(mod_name, file_name)
        sys.modules[mod_name] = m
        return m

    raise ImportError("...")
```

Implementation (2)

In [178]:

```
from threading import Lock

class SingletonFinal(type):
    instance = None
    lock = Lock()
    def __call__(cls, *args, **kw):
        with cls.lock:
            if not cls.instance:
                cls.instance = super(SingletonFinal, cls).__call__(*args, **kw)
        return cls.instance

    def __init__(cls, name, bases, namespace):
        super(SingletonFinal, cls).__init__(name, bases, namespace)
        for klass in bases:
            if isinstance(klass, SingletonFinal):
                print "**debug** ", name, bases, namespace
                raise TypeError(str(klass.__name__) + " is final")
```

In [52]:

```
class ASingleton(object):
    __metaclass__ = SingletonFinal

a = ASingleton()
b = ASingleton()

assert a is b
print(a.__class__.__name__, b.__class__.__name__)
```

```
('ASingleton', 'ASingleton')
```

3.5. MonoState (Other)

Also known as

Borg

Intent

Enforces a behaviour like sharing the same state amongst all instances.

Applicability

Use the Monostate pattern when

- The same state must be shared across all instances of a class.
- **Typically this pattern might be used everywhere a Singleton might be used. Singleton usage however is not transparent, Monostate usage is.**
- Monostate has one major advantage over singleton. **The subclasses might decorate the shared state as they wish and hence can provide dynamically different behaviour than the base class.**

Implementation

In [4]:

```
class Borg(object):
    _shared_state = {}

    def __new__(cls, *args, **kwargs):
        obj = super(Borg, cls).__new__(cls, *args, **kwargs)
        obj.__dict__ = cls._shared_state
        return obj


class Foo(Borg): pass
class Bar(Foo): pass

b1, b2, f1 = Borg(), Borg(), Foo()

b1.a = 10
b2.b = 20
print f1.a, f1.b
```

```
10 20
```

3.6. Prototype

Intent

Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Applicability

Use the Prototype pattern

- when the classes to instantiate are specified **at run-time**, for example, by dynamic loading; or
- when instances of a class can have one of only a few different combinations of state. It may be **more convenient** to install a corresponding number of prototypes and clone them **rather than instantiating the class manually**, each time with the appropriate state

Implementation

- normally use copy to do so

In [ ]:

```
import copy
class SubComponent(object):
    pass

class ComplexData(object):
    def __init__(self):
        self.d1 = "abc"
        self.d2 = [1,2,3]
        self.d3 = SubComponent()

    # make it always deep copy
#     def __copy__(self,):
#         return copy.deepcopy(self)
```

In [ ]:

```
c = ComplexData()
copy1 = copy.copy(c)
copy2 = copy.deepcopy(c)
del c.d2[0]
```

In [176]:
```
print c.d1, c.d2, c.d3
print copy1.d1, copy1.d2, copy1.d3
print copy2.d1, copy2.d2, copy2.d3
```

```
abc [2, 3] <__main__.SubComponent object at 0x104770650>
abc [2, 3] <__main__.SubComponent object at 0x104770650>
abc [1, 2, 3] <__main__.SubComponent object at 0x1047703d0>
```

4. Structural Design Patterns Practice

- Deal with the mutual composition of classes or objects
- GOF Covered:
  - Decorator
  - Proxy
  - Bridge
- **Functional Programming** related:
  - **Currying**
  - **Compose**
- Others
  - Mixin

4.1. Decorator

Intent

wrap a function or class to add more function or twist the structure/behaviours and keep the original signatures

Use Python **decorator syntax** rather than **inheritance**

Implementation

In [2]:
```
import wrapt

@wrapt.decorator
def log(fn, instance, args, kwargs):
    print '"{}({})" enter'.format(fn.func_name, ','.join(args))
    ret = fn(*args, **kwargs)
    print '"{}({})" exit.'.format(fn.func_name, ','.join(args))
    return ret
```

In [3]:
```
@log
def convert(s1):
    return int(s1)

convert("123")
```

```
"convert(123)" enter
"convert(123)" exit.
```

Out[3]:
```
123
```

Another Usage

In [4]:
```
def convert(s1):
    return int(s1)

logged_convert = log(convert)
logged_convert("345")
```

```
"convert(345)" enter
"convert(345)" exit.
```

Out[4]:
```
345
```

Note

- **Actually in Python, you can do much more things with Decorator**
- For more **cool** things you can do with **decorator**, refer to my talk Python Hooking, Patching and Injection (https://github.com/wjo1212/ChinaPyCon2016) in PyCon 2016 Shenzhen

4.2. Proxy

Also known as

Surrogate

Intent

Provide a surrogate or placeholder for another object to control access to it.

Applicability

Proxy is applicable when:

- a **remote proxy** provides a local representative for an object in a different address space.
- a virtual proxy creates **expensive objects on demand.**
- a protection proxy **controls access** to the original object.

Implementation

In [ ]:

```
# Note: consider a 10GB table
students = AzureTable('user1.storage.azure.com/table/students')

# Only get the data when be accessed
print students.xiaoMing.Address.data
print students.Jim.birthday.data
```

In [180]:
```
class AzureTable(object):
    def __init__(self, url, row=None, col=None, level='table'):
        self.__url = url
        self.__row = row
        self.__col = col
        self.__level = level

    def _fetch(self):
        assert self.__level == 'col'
        print '*** Downloading from\n\t"{}/{}/{}"'.format(self.__url,
                                                          self.__row,
                                                          self.__col)
        print '*** Downloading Complete'
        return "Hello PyCon 2016: {}.{}".format(self.__row, self.__col)

    def __getattr__(self, item):
        if self.__level == 'table':
            return AzureTable(self.__url, row=item, level='row')
        elif self.__level == 'row':
            return AzureTable(self.__url, row=self.__row, col=item, level='col')

        if item == 'data':
            return self._fetch()
```

In [181]:
```
students = AzureTable('user1.storage.azure.com/table/students')

print students.xiaoMing.Address.data
print "-" * 30
print students.Jim.birthday.data
```

```
*** Downloading from
	"user1.storage.azure.com/table/students/xiaoMing/Address"
*** Downloading Complete
Hello PyCon 2016: xiaoMing.Address
------------------------------
*** Downloading from
	"user1.storage.azure.com/table/students/Jim/birthday"
*** Downloading Complete
Hello PyCon 2016: Jim.birthday
```

Note

Bridge pattern could also consider use __getattr__ to provide dynamic bridged object as well if need.

For more mechanism about **property access hooking**, Refer to my talk Python Hooking, Patching and Injection (https://github.com/wjo1212/ChinaPyCon2016) in PyCon 2016 Shenzhen

Another example in system module: weakref https://pymotw.com/2/weakref/index.html (https://pymotw.com/2/weakref/index.html)

4.3. Currying (FP)

Intent

Currying provides a way for working with functions that take multiple arguments, and using them in frameworks where functions might take only one argument.

Somehow an **Adapter for functional programing**

Implementation (functools.partial)

In [183]:
```
from functools import partial

int16 = partial(int, base=16)
int16("FF")
```

Out[183]: 255

Implementation (currying)

In [103]:
```
from toolz.functoolz import curry

@curry
def sum5(a, b, c, d, e):
    return a + b + c + d + e

assert 15 == sum5(1)(2)(3)(4)(5)
assert 15 == sum5(1, 2, 3)(4, 5)
assert 15 == sum5(1, 2, 3, 4, 5)
```

4.4. Compose (FP)

Intend

compose high level function on top of low level components

Examples:

How to sum of all even number inside the string?

s1 = "12x3y45z67t89” => 2+4+6+8 => 20

In [107]:
```
s1 = "12x3y45z67t89"

sum(int(x) for x in s1 if x.isdigit() and int(x) % 2 == 0)
```

Out[107]:
```
20
```

Implementation

In [108]:

```
from toolz.functoolz import compose
from functools import partial
sum_even_from_str = compose(sum, partial(filter, lambda _:_%2== 0), partial(map, int), partial(filt
```

In [112]:
```
print sum_even_from_str(s1)
print sum_even_from_str("6x 4x 2x 1x1")
```

```
20
12
```

In [114]:
```
from fn import F, _
sum_even_from_str = F(filter, str.isdigit) >> F(map, int) >> F(filter, _%2==0) >> sum
```

In [115]:
```
print sum_even_from_str(s1)
print sum_even_from_str("6x 4x 2x 1x1")
```

```
20
12
```

4.5. Mixin (Other)

Intent

Add (`mix-in`) new features into a class, object statically or dynamically, normally via meta-programming and/or monkey patching technology, but not limited to.

For more mechanism about **metaclass** and **monkey patch**, Refer to my talk Python Hooking, Patching and Injection (https://github.com/wjo1212/ChinaPyCon2016) in PyCon 2016 Shenzhen

Implementation

In [7]:
```python
class Dancer(object):
    def dance(self,):
        print '"{}" is dancing'.format(self.name)


class Student(object):
    def __init__(self, name):
        self.name = name

    def study(self,):
        print '"{}" is studying'.format(self.name)
```

In [8]:
```python
s1 = Student('s1')
s1.study()
# s1.dance()   # cannot dance
print "-" * 30

Student = type('Student', (Dancer,), dict(Student.__dict__))

s2 = Student('s2')
s2.study()
s2.dance()   # any new object can dance

print "-" * 30
s1.__class__ = Student # patch s1
s1.dance()  # student now can dance too
```

```
"s1" is studying
------------------------------
"s2" is studying
"s2" is dancing
------------------------------
"s1" is dancing
```

5. Behavioral Design Patterns Practice

- Analyze the ways in which classes or objects interact and distribute responsibilities among them
- **GOF Covered:**

  - Template Method
  - Strategy, Observer
  - Iterator, Visitor, Builder (*)
- **Functional Programming** related:
  - **High Order Function**
  - Memorization
  - **Monads**
- **Others:**
  - **Double Dispatch**
  - Mock-up

5.1. Template Method

Also Known as

Self-delegation

Intent

Define the **skeleton of an algorithm** in an operation, **deferring some steps to subclasses.** Template method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Applicability

The Template Method pattern should be used to implement the invariant parts of an algorithm **once** and leave it up to **subclasses to implement the behavior that can vary**

- when common behavior among subclasses should be factored and localized in a common class to **avoid code duplication.**
- to control subclasses extensions. You can define a template method that calls "hook" operations at specific points, thereby permitting extensions only at those points

Use cases

- Test Cases

Note

Be more flexible and powerful in dynamic lanugage with more powerful inspection and meta-programming capability

5.1.1. Classic Implementation

In [14]:
```
from abc import abstractmethod, ABCMeta

class AbstractTestCase(object):
    __metaclass__ = ABCMeta

    def setup(self,):
        pass

    def teardown(self,):
        pass

    @abstractmethod
    def test(self,):
        pass

    def run(self,):
        self.setup()
        self.test()
        self.teardown()
```

In [15]:
```
class Test1(AbstractTestCase):
    def setup(self,):
        print "prepare"

    def teardown(self,):
        print 'teardown'

    def test(self,):
        print "do some test"
        assert 1 == int('1')


t = Test1()
t.run()
```

```
prepare
do some test
teardown
```

5.1.2. More Flexible Implementation (using tag)

In [26]:
```
from types import MethodType

case_tag_list = []
def case_tag(fn):
    case_tag_list.append(fn)
    return fn

class TestCaseTemplate(object):
    def __init__(self):
        self.run_list = []
        for x in dir(self):
            fn = getattr(self, x)
            if isinstance(fn, MethodType) and fn.im_func in case_tag_list:
                self.run_list.append(fn)

    def setup(self,): pass
    def teardown(self,): pass

    def run(self,):
        self.setup()
        for fn in self.run_list:
            fn()
        self.teardown()
```

In [27]:
```
class Test1(TestCaseTemplate):
    def setup(self,):
        print "prepare"

    def teardown(self,):
        print 'teardown'

    @case_tag
    def case1(self,):
        print "do some test1"
        assert 1 == int('1')

    @case_tag
    def case2(self,):
        print "do some test2"
        assert 1 == int('1')

t = Test1()
t.run()
```

```
prepare
do some test1
do some test2
teardown
```

5.1.3. More Flexible Implementation (w/o tag)

In [9]:
```
from types import MethodType

class TestCaseTemplate(object):
    def __init__(self):
        self.run_list = []
        for x in dir(self):
            fn = getattr(self, x)
            if isinstance(fn, MethodType) \
                and fn.func_name.startswith('test'):
                self.run_list.append(fn)

    def setup(self,): pass

    def teardown(self,): pass

    def run(self,):
        self.setup()
        for fn in self.run_list:
            fn()
        self.teardown()
```

In [31]:
```
class Test1(TestCaseTemplate):
    def setup(self,):
        print "prepare"

    def teardown(self,):
        print 'teardown'

    def test1(self,):
        print "do some test1"
        assert 1 == int('1')

    def test2(self,):
        print "do some test2"
        assert 1 == int('1')

t = Test1()
t.run()
```

```
prepare
do some test1
do some test2
teardown
```

5.2. Strategy

Also known as

Policy

Intent

Define **a family of algorithms**, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Applicability

Use the Strategy pattern when

- **many related classes differ only in their behavior.**
- you need **different variants of an algorithm.**

5.2.1. Classic implementation

In [104]:

```
import json
import re
from abc import ABCMeta, abstractmethod

class IDataParser(object):
    __metaclass__ = ABCMeta

    @abstractmethod
    def parse(cls, content): pass

class IDataLoader(object):
    __metaclass__ = ABCMeta

    @abstractmethod
    def get_content(self): pass

class ConfigData(object):
    def __init__(self, loader, parser):
        assert isinstance(loader, IDataLoader) and isinstance(parser, IDataParser)
        self.data = parser.parse(loader.get_content())

    def __getitem__(self, item):
        return self.data.get(item, None)
```

In [105]:

```
class JsonParser(IDataParser):
    def parse(cls, content):
        return json.loads(content)

class KvParser(IDataParser):
    def parse(cls, content):
        return dict(re.findall(r'(\w+)\s*=\s*"?([\w\s]+)"?', content))

class S3Storage(IDataLoader):
    def get_content(self):
        # read content from AWS S3
        return '{ "d1": "s3 d1", "d2": "s3 d2" }'

class FileStorage(IDataLoader):
    def get_content(self):
        # read content from file path
        return 'd1="file d1", d2="file d2"'
```

In [106]:

```
s3_config = ConfigData(S3Storage(), JsonParser())
print s3_config['d1']
print s3_config['d2']

file_config = ConfigData(FileStorage(), KvParser())
print file_config['d1']
print file_config['d2']
```

```
s3 d1
s3 d2
file d1
file d2
```

5.2.2. implementation with duck typing

In [107]:

```
class ConfigData(object):
    def __init__(self, loader, parser):
        self.data = parser.parse(loader.get_content())

    def __getitem__(self, item):
        return self.data.get(item, None)

class JsonParser(object):
    def parse(cls, content):
        return json.loads(content)

class KvParser(object):
    def parse(cls, content):
        return dict(re.findall(r'(\w+)\s*=\s*"?([\w\s]+)"?', content))

class S3Storage(object):
    def get_content(self):
        # read content from AWS S3
        return '{ "d1": "s3 d1", "d2": "s3 d2" }'

class FileStorage(object):
    def get_content(self):
        # read content from file path
        return 'd1="file d1", d2="file d2"'
```

In [109]:

```
s3_config = ConfigData(S3Storage(), JsonParser())
print s3_config['d1']
print s3_config['d2']

file_config = ConfigData(FileStorage(), KvParser())
print file_config['d1']
print file_config['d2']
```

```
s3 d1
s3 d2
file d1
file d2
```

5.2.3. Simpler way using functions

In [110]:
```
import json
import re

def parse_json(content):
    return json.loads(content)

def parse_kv(content):
    return dict(re.findall(r'(\w+)\s*=\s*"?([\w\s]+)"?', content))

def load_s3():
        # read content from AWS S3
        return '{ "d1": "s3 d1", "d2": "s3 d2" }'

def load_file():
        # read content from file path
        return 'd1="file d1", d2="file d2"'
```

In [111]:
```
s3_config = parse_json(load_s3())
print s3_config['d1']
print s3_config['d2']

file_config = parse_kv(load_file())
print file_config['d1']
print file_config['d2']
```

```
s3 d1
s3 d2
file d1
file d2
```

5.3. Observer

Also known as

Dependents, Publish-Subscribe

Intent

Define a **one-to-many dependency** so that when **one object changes state**, all its dependents are **notified** and updated **automatically**.

Applicability

- when an abstraction has **two aspects, one dependent on the other.**
- when a change to one object requires changing others, and you don't know how many objects need to be changed
- when an object should be able to notify other objects without knowing who these objects are.

5.3.1. Class Level Implementation

In [11]:

```
import wrapt

class Data(object):
    def __init__(self):
        self.d1 = "1"
        self.d2 = "hello world"

    def change1(self,):
        self.d1 = "123"

    def chang2(self, new_val):
        self.d2 = new_val
```

In [12]:
```
@wrapt.decorator
def ob1(fn, instance, *args):
    attr, val = args[0]
    ret = fn(attr, val)
    print "ob1: ", attr, " is changed to: ", val
    return ret

@wrapt.decorator
def ob2(fn, instance, *args):
    attr, val = args[0]
    ret = fn(attr, val)
    print "ob2: ", attr, " is changed to: ", val
    return ret


def add_ob(cls, ob):
    cls.__setattr__ = ob(cls.__setattr__)

add_ob(Data, ob1)
add_ob(Data, ob2)
```

In [14]:
```
x, y = Data(), Data()

x.d1 = "abc"
x.d2 = "hello data"
x.d3 = 123

y.d1 = "xxx"
# Question: how to observe at object level ?
```

```
ob1:  d1  is changed to:  1
ob2:  d1  is changed to:  1
ob1:  d2  is changed to:  hello world
ob2:  d2  is changed to:  hello world
ob1:  d1  is changed to:  1
ob2:  d1  is changed to:  1
ob1:  d2  is changed to:  hello world
ob2:  d2  is changed to:  hello world
ob1:  d1  is changed to:  abc
ob2:  d1  is changed to:  abc
ob1:  d2  is changed to:  hello data
ob2:  d2  is changed to:  hello data
ob1:  d3  is changed to:  123
ob2:  d3  is changed to:  123
ob1:  d1  is changed to:  xxx
ob2:  d1  is changed to:  xxx
```

5.3.2. Object Level Implmentation

In [ ]:

```
import wrapt

class Data(object):
    def __init__(self):
        self.d1 = "1"
        self.d2 = "hello world"

    def change1(self,):
        self.d1 = "123"

    def chang2(self, new_val):
        self.d2 = new_val
```

In [1]:

```
def ob1(fn, instance, *args):
    attr, val = args[0]
    ret = fn(attr, val)
    print "ob1: ", attr, " is changed to: ", val
    return ret

def ob2(fn, instance, *args):
    attr, val = args[0]
    ret = fn(attr, val)
    print "ob2: ", attr, " is changed to: ", val
    return ret


def add_ob(obj, ob):
    @wrapt.decorator
    def bind_obj(fn, instance, *args):
        if instance is obj:
            return ob(fn, instance, *args)
        else:
            return fn(*args[0], **args[1])

    type(obj).__setattr__ = bind_obj(type(obj).__setattr__)
```

In [2]:
```
d1, d2 = Data(), Data()

add_ob(d1, ob1)
add_ob(d1, ob2)
add_ob(d2, ob1)
```

In [7]:
```
d1.d1 = "abc"
d2.d1 = 123
d1.d2 = 232
d2.d3 = 1111
d1.d3 = 'xxx'
```

```
ob1:  d1  is changed to:  abc
ob2:  d1  is changed to:  abc
ob1:  d1  is changed to:  123
ob1:  d2  is changed to:  232
ob2:  d2  is changed to:  232
ob1:  d3  is changed to:  1111
ob1:  d3  is changed to:  xxx
ob2:  d3  is changed to:  xxx
```

5.3.3. Event Level Implementation (C# Style)

Event handler in C# Code

In [ ]:
```
using System;
class Observable {
    public event EventHandler SomethingHappened;

    public void DoSomething() {
        EventHandler handler = SomethingHappened;
        if (handler != null) {
            handler(this, EventArgs.Empty);
        }
    }
}
class Observer {
    public void HandleEvent(object sender, EventArgs args) {
        Console.WriteLine("Something happened to " + sender);
    }
}
class Test {
    static void Main() {
        Observable observable = new Observable();
        Observer observer = new Observer();
        observable.SomethingHappened += observer.HandleEvent;

        observable.DoSomething();
    }
}
```

Implementation

In [209]:
```
from events import Events

class Observable(object):
    __events__ = ('on_name_change', 'on_value_change')
    events = Events()

    def do_something(self,):
        self.events.on_name_change('new data')

class Observer:
    def handle_event(self, new_data):
        print "data is changed: ", new_data


observable = Observable()
observer = Observer()

observable.events.on_name_change += observer.handle_event
observable.do_something()
```

```
data is changed:  new data
```

5.4. Iterator

we will skip it

it's so popular in Python and functional programming

refer to map, filter, reduce, itertools., *toolz.itertoolz.* etc

5.5. High Order Function (FP)

Intent

A function that accept function as parameters or return function

Note

a kind of **Strategy Pattern for Functional Programming***

Implementation

In [14]:
```
def my_reduce(function, sequence, initial=None):
    it = iter(sequence)
    ret = initial or it.next()

    for x in it:
        ret = function(ret, x)

    return ret
```

In [17]:
```
from functools import partial
my_sum = partial(my_reduce, lambda x,y: x+y, initial=0)
my_pdt = partial(my_reduce, lambda x,y: x*y, initial=1)
my_len = partial(my_reduce, lambda x,y: x+1, initial=0)
```

In [18]:
```
print my_sum([1,2,3,4])   # 1 + 2 + 3 + 4
print my_pdt([1,2,3,4])   # 1 * 2 * 3 * 4
print my_len([1,2,3,4])   # 1 + 1 + 1 + 1
```

```
10
24
4
```

5.6. Memorization (FP)

Also Known as

Cache

Intent

To avoid expensive re-acquisition of resources by not releasing the resources immediately after their use. The resources retain their identity, are kept in some fast-access storage, and are re-used to avoid having to acquire them again.

Applicability

Use the Caching pattern(s) when

Implementation

In [76]:
```
from toolz.functoolz import memoize
import urllib2

@memoize
def download(path):
    print "start download: ", path
    f = urllib2.urlopen(path)
    ret = f.read()
    f.close()
    return ret

p1 = 'http://localhost:8888/static/base/images/logo.png'
p2 = 'http://localhost:8888/kernelspecs/python2/logo-64x64.png'

print len(download(p1))
print len(download(p1))
print len(download(p2))
print len(download(p2))
```

```
start download:  http://localhost:8888/static/base/images/logo.png (http://localhost:8888/static/base/images/logo.png)
4473
4473
start download:  http://localhost:8888/kernelspecs/python2/logo-64x64.png (http://localhost:8888/kernelspecs/python2/logo-64x64.png)
2180
2180
```

5.7. Double Dispatch (Other)

Also Known As

Multimethod

Intent

Double Dispatch pattern is a way to create maintainable dynamic behavior based on receiver and parameter types.

Applicability

Use the Double Dispatch pattern when

- the dynamic behavior is not defined only based on receiving object's type but also on the receiving method's parameter

Note

Language supporting double dispatch normally doesn't need **Visitor Pattern**

Double Dispatch in Java

In [ ]:
```
class Fruit{ }
class Apple extends Fruit{ }
class Banana extends Fruit{  }

class People {
    public void eat(Fruit f) { System.out.println("People eat Fruit"); }
    public void eat(Apple f) { System.out.println("People eat Apple"); }
    public void eat(Banana f) { System.out.println("People eat Banana"); }
}

class Boy extends People {
    public void eat(Fruit f) { System.out.println("Boy eats Fruit"); }
    public void eat(Apple f) { System.out.println("Boy eats Apple"); }
    public void eat(Banana f) { System.out.println("Boy eats Banana"); }
}

public class multipatch {
    public static void main(String[] argu) {
        People boy = new Boy();
        Fruit apple = new Apple();
        Fruit banana = new Banana();
        boy.eat(apple);   // Boy eats Fruit
        boy.eat(banana);  // Boy eats Fruit
    }
}
```

Implementation

Also cover Builder Pattern here

In [36]:

```
from abc import ABCMeta, abstractmethod
from functools import partial
from multipledispatch import dispatch

disptch = partial(dispatch, namespace=dict())

class DocElement(object):
    def __str__(self): return "Doc Element"

class PlainText(DocElement):
    def __str__(self): return "Plain Text"

class RichText(PlainText):
    def __str__(self): return "Apple"

class Image(DocElement):
    def __str__(self): return "Image"

class GifImage(Image):
    def __str__(self): return "Gif Image"
```

In [40]:
```
class Builder(object):
    __metaclass__ = ABCMeta

    def __repr__(self):
        return ''

    @abstractmethod
    @dispatch(DocElement)
    def make(self, element): pass

    @abstractmethod
    @dispatch(PlainText)
    def make(self, element): pass

    @abstractmethod
    @dispatch(RichText)
    def make(self, element): pass

    @abstractmethod
    @dispatch(Image)
    def make(self, element): pass

    @abstractmethod
    @dispatch(GifImage)
    def make(self, element): pass
```

In [41]:

```
class RtfBuilder(Builder):
    @dispatch(DocElement)
    def make(self, element):
        print "RTF make: ", element
        return self

    @dispatch(PlainText)
    def make(self, element):
        print "RTF make: ", element
        return self

    @dispatch(RichText)
    def make(self, element):
        print "RTF make: ", element
        return self

    @dispatch(Image)
    def make(self, element):
        print "RTF make: ", element
        return self

    @dispatch(GifImage)
    def make(self, element):
        print "RTF make: ", element
        return self
```

In [42]:
```
class DocxBuilder(Builder):
    @dispatch(DocElement)
    def make(self, element):
        print "DOCX make: ", element
        return self

    @dispatch(PlainText)
    def make(self, element):
        print "DOCX make: ", element
        return self

    @dispatch(RichText)
    def make(self, element):
        print "DOCX make: ", element
        return self

    @dispatch(Image)
    def make(self, element):
        print "DOCX make: ", element
        return self

    @dispatch(GifImage)
    def make(self, element):
        print "DOCX make: ", element
        return self
```

In [43]:

```
txt = PlainText()
img = Image()
gif = GifImage()
rtxt = RichText()

builder1 = RtfBuilder()
builder2 = DocxBuilder()

builder1.make(txt).make(img).make(gif).make(rtxt)
print '-' * 30
builder2.make(txt).make(img).make(gif).make(rtxt)
```

```
RTF make:  Plain Text
RTF make:  Image
RTF make:  Gif Image
RTF make:  Apple
------------------------------
DOCX make:  Plain Text
DOCX make:  Image
DOCX make:  Gif Image
DOCX make:  Apple
```

Out[43]:

5.8. Monads (FP)

Intent

Monad pattern based on monad from linear algebra represents the way of **chaining operations together step by step.** Binding functions can be described as passing one's output to another's input basing on the **'same type' contract.** Formally, monad consists of a type constructor M and two operations: bind - that takes monadic object and a function from plain object to monadic value and returns monadic value return - that takes plain type object and returns this object wrapped in a monadic value.

Applicability

Use the Monad in any of the following situations

- when you want to chain operations easily
- when you want to apply each function regardless of the result of any of them

Example

In [63]:
```
def sqrt(x):
    if x < 0:
        return None
    else:
        return x ** 0.5

def divide100(divisor):
    if divisor == 0:
        return None
    else:
        return 100 / divisor
```

In [64]:
```
from fn import F
cal = F() >> sqrt >> divide100 >> sqrt >> divide100
```

error happens during fluent API calling

In [65]:
```
print cal(3)
try:
    print cal(0)
except Exception as ex:
    print ex
```

```
13.1607401295
unsupported operand type(s) for /: 'int' and 'NoneType'
```

Implementation

In [66]:
```
import wrapt
from fn import F

@wrapt.decorator
def my_shift(fn, instance, args, kwargs):

    def handle_none(param,):
        if param is None:
            return None

        return args[0](param)

    return fn(handle_none)


F.__rshift__ = my_shift(F.__rshift__)
F.__lshift__ = my_shift(F.__lshift__)
```

In [67]:
```
cal = F() >> sqrt >> divide100 >> sqrt >> divide100

print cal(3)
print cal(0) # can handle error and output None
```

```
13.1607401295
None
```

5.9. Mock-up (Other)

Intent

Mock or patching a module or API with stub or predefined class/objects or methods for specific purpose like testing, normally uses Monkey Patch technology.

Example

In [ ]:
```
# %load code/mut.py
import conf_loader as cl

def logic():
    config = cl.load()  # load settings from some REST

    # do a lot of complex settings

    return "abc"
```

In [58]:
```
import code.mut as mut

def test_case_1():
    assert mut.logic() == "abc", "test failed"
    print "test passed"

test_case_1()
```

```
** conf_loader: load data from remote system....
test passed
```

what if the conf_loader.load:

- not available (complex system)
- hard to configure test data
- slow as heavy operation

In [ ]:
```
# %load code/conf_loader.py
def load():
    print "** conf_loader: load data from remote system...."

    # raise ValueError("time-out: remote system no response")

    # takes long time to get data
    return "...."
```

In [60]:
```
import mock
import code.mut as mut

@mock.patch("code.conf_loader.load")
def main_v1(mock_load):
    mock_load.return_value = '{"settings":"..."}'

    assert mut.logic() == "abc", "test failed"
    print "test passed"

main_v1()
```

```
test passed
```

6. Concurrent Patterns

- Deal with the multi-threaded (or multi-process) programming paradigm, concerns the performance and effective ways among class/objects in a concurrent context.
- **Patterns Covered**:
    - **Reader Writer Lock**
    - Guarded suspension
    - **Advanced Producer and Consumer**
    - Promise (Future)
    - Thread Pool
    - **Executor Service**

6.1. Reader Writer Lock

Also Known as

shared-exclusive lock

Intent:

Shared resources that allow multiple readers to read in parallel but exclusive for only one writer to write.

Applicability:

Application need to **increase the performance** of resource synchronize for multiple thread, in particularly there are **mixed read/write operations**.

Traddtional Lock (relatively bad performance)

In [44]:

```
from threading import Lock, Thread
from contextlib import contextmanager
import time
import itertools as it

class FakeReadWriteLock(object):
    def __init__(self):
        self.__monitor = Lock()

    def read(self):
        return self.__monitor

    def write(self):
        return self.__monitor
```

In [45]:

```
def run_read(rwl, data):
    for x in range(3000):
        with rwl.read():
            time.sleep(0.0001) # suppose hold lock for 0.1 ms
            data[-1]  # read data

def run_write(rwl, data):
    for x in range(3000):
        with rwl.write():
            time.sleep(0.0002)  # suppose hold lock for 0.2 ms
            data.append(x) # write data

def test(rwl):
    data = [-1]
    read_ths = [Thread(target=run_read, args=(rwl, data))
                for x in range(20)]
    write_ths = [Thread(target=run_write, args=(rwl, data))
                 for x in range(2)]
    s = time.time()
    for t in it.chain(write_ths, read_ths):
        t.start()
    for t in it.chain(write_ths, read_ths):
        t.join()
    e = time.time()
    print "exit: time used = {0:.3} seconds".format(e-s)
```

In [46]:

```
rwl = FakeReadWriteLock()
test(rwl)
```

```
exit: time used = 10.2 seconds
```

Implementation

In [48]:

```
class ReadWriteLock(object):
    def __init__(self):
        self.__monitor = Lock()
        self.__exclude = Lock()
        self.readers = 0

    @contextmanager
    def read(self):
        with self.__monitor:
            self.readers += 1
            if self.readers == 1:
                self.__exclude.acquire()
        yield self
        with self.__monitor:
            self.readers -= 1
            if self.readers == 0:
                self.__exclude.release()

    def write(self):
        return self.__exclude
```

In [49]:

```
rwl = ReadWriteLock()
test(rwl)
```

```
exit: time used = 3.79 seconds
```

Further question:

how to support write-preferring to prevent writer Starvation of writing (when read locks for too much time)?

6.2. Guarded suspension

Intent

guarded suspension is a software design pattern for managing operations that require **both a lock to be acquired and a precondition to be satisfied** before the operation can be executed.

Applicability

The guarded suspension pattern is typically applied to method calls in object-oriented programs, and involves suspending the method call, and the calling thread, **until the precondition (acting as a guard) is satisfied.**

Examples (in Java)

In [ ]:
```
# https://en.wikipedia.org/wiki/Guarded_suspension
public class Example {
    synchronized void guardedMethod() {
        while (!preCondition()) {
            try {
                // Continue to wait
                wait();
                // …
            } catch (InterruptedException e) {
                // …
            }
        }
        // Actual task implementation
    }
    synchronized void alterObjectStateMethod() {
        // Change the object state
        // …
        // Inform waiting threads
        notify();
    }
}
```

Note: we will see implementation with next Pattern

6.3. Producer and Consumer

Intent

Producer Consumer Design pattern is a classic concurrency pattern which reduces coupling between Producer and Consumer by separating Identification of work with Execution of Work.

Applicability

Use the Producer Consumer pattern when

- decouple system by separate work in two process produce and consume.
- addresses the issue of different timing require to produce work or consuming work

Implementaitons (PaC w/Guarded suspension)

In [58]:

```
import threading as td
from threading import *
import time

class Queue(object):
    data = []
    item_id = 1
    def __init__(self, capability = 20): self.max_lst_size = capability

    @property
    def data_count(self): return len(self.data)
    @property
    def vacancy_count(self): return self.max_lst_size - len(self.data)

    def consume(self, size):
        item = self.data[:size]
        del self.data[:size]
        return item

    def produce(self, size):
        item  = range(self.item_id, self.item_id + size)
        self.data.extend(item)
        self.item_id += size
        return item
```

In [76]:

```
g_exit = False

def consumer(cv_consume, cv_produce, q, capability):
    def meet_condition():
        return q.data_count >= capability

    def consume_item():
        items = q.consume(capability)
        print 'consumer ', td.current_thread().name, 'consumes item', items

    def need_exit():
        return g_exit

    # Consume one item
    while True:
        with cv_consume:
            while not meet_condition() and not need_exit():
                cv_consume.wait()

            if need_exit():
                print 'consumer ', td.current_thread().name, 'exit'
                break

            consume_item()
            cv_produce.notify_all()
```

In [77]:
```
def producer(cv_consume, cv_produce, q, capability):
    def meet_condition():
        return q.vacancy_count >= capability

    def produce_item():
        item = q.produce(capability)
        print 'producer ', td.current_thread().name, 'procudes item', item

    def need_exit():
        return g_exit

    # Consume one item
    while True:
        with cv_produce:
            while not meet_condition() and not need_exit():
                cv_produce.wait()

            if need_exit():
                print 'producer ', td.current_thread().name, 'exit'
                break

            produce_item()
            cv_consume.notify_all()
```

In [87]:
```
def main():
    global g_exit
    g_exit = False

    l, q = RLock(), Queue()
    cv_consume, cv_produce = Condition(lock=l), Condition(lock=l)

    c1 = Thread(target=consumer, args=(cv_consume, cv_produce, q, 2))
    c2 = Thread(target=consumer, args=(cv_consume, cv_produce, q, 7))
    p1 = Thread(target=producer, args=(cv_consume, cv_produce, q, 5))
    p2 = Thread(target=producer, args=(cv_consume, cv_produce, q, 3))

    lst = [c1, c2, p1, p2]
    for p in lst: p.start()

    # keep running for 1 ms, then exit and wait up all threads
    time.sleep(0.001)
    g_exit = True

    with cv_consume, cv_produce:
        cv_produce.notify_all()
        cv_consume.notify_all()

    for p in lst: p.join()

    print "exist main, q size =", q.data_count
```

Result

In [88]:
```
main()
```

```
consumer  Thread-80 consumes item [45, 46]
consumer  Thread-80 consumes item [47, 48]
consumer  Thread-80 consumes item [49, 50]
consumer  Thread-80 consumes item [51, 52]
producer  Thread-82 procudes item [1, 2, 3, 4, 5]
producer  Thread-82 procudes item [6, 7, 8, 9, 10]
consumer  Thread-80 consumes item [53, 1]
consumer  Thread-80 consumes item [2, 3]
consumer  Thread-80 consumes item [4, 5]
consumer  Thread-80 consumes item [6, 7]
producer  Thread-83 procudes item [11, 12, 13]
producer  Thread-82 procudes item [14, 15, 16, 17, 18]
consumer  Thread-80 consumes item [8, 9]
producer  Thread-83 procudes item [19, 20, 21]
producer  Thread-83 procudes item [22, 23, 24]
producer  Thread-82 procudes item [25, 26, 27, 28, 29]
consumer  Thread-80 consumes item [10, 11]
consumer  Thread-80 consumes item [12, 13]
producer  Thread-83 procudes item [30, 31, 32]
producer  Thread-82 exit
consumer  Thread-81 exit
consumer  Thread-80 exit
producer  Thread-83 exit
exist main, q size = 19
```

6.4. Promise (Future)

Also known as

CompletableFuture

Specifically, when usage is distinguished, a **future** is a read-only placeholder view of a variable, while a **promise** is a writable, single assignment container which sets the value of the future.

https://en.wikipedia.org/wiki/Futures_and_promises (https://en.wikipedia.org/wiki/Futures_and_promises)

Intent

A Promise represents a proxy for a value not necessarily known when the promise is created. It allows you to associate dependent promises to an asynchronous action's eventual success value or failure reason. Promises are a way to write async code that still appears as though it is executing in a synchronous way.

Applicability

Promise pattern is applicable in concurrent programming when some work needs to be done asynchronously and:

- code maintainablity and readability suffers due to callback hell.
- you need to compose promises and need better error handling for asynchronous tasks.
- you want to use functional style of programming.

Implementation - Implicitly (with Executor Service)

will see with later Pattern

Implementation - Explicitly

In [1]:
```python
from concurrent.futures import Future
```

In [2]:
```python
# In thread (or process) A
f = Future()
print f.done()
print f.running()
```

```
False
False
```

In [99]:
```
# In thread (or process) B
f.set_result(100)

# In thread (or process) A
print f.result()
print f.exception()
```

```
100
None
```

Transfer exception

In [9]:
```
# Inthread (or process) B
f.set_exception(ZeroDivisionError("some error happens"))
```

In [11]:
```
# In another thread (or process)
#print f.result()  # will drectly throws the exception
print f.exception()
```

```
some error happens
```

6.5. Thread Pool and Executor Service

Intent

It is often the case that tasks to be executed are short-lived and the number of tasks is large. Creating a new thread for each task would make the system spend more time creating and destroying the threads than executing the actual tasks. Thread Pool solves this problem by reusing existing threads and eliminating the latency of creating new threads.

Applicability

Use the Thread Pool pattern when

you have a large number of short-lived tasks to be executed in parallel

Implementation

In [ ]:
```
from concurrent import futures
import math

PRIMES = [
    112272535095293,
    112582705942171,
    112272535095293,
    115280095190773,
    115797848077099,
    1099726899285419]
```

In [13]:
```
def is_prime(n):
    if n % 2 == 0:
        return False

    sqrt_n = int(math.floor(math.sqrt(n)))
    for i in range(3, sqrt_n + 1, 2):
        if n % i == 0:
            return False
    return True

def main():
    with futures.ProcessPoolExecutor() as executor:
        for number, prime in zip(PRIMES, executor.map(is_prime,
                                                      PRIMES)):
            print('%d is prime: %s' % (number, prime))

main()
```

```
112272535095293 is prime: True
112582705942171 is prime: True
112272535095293 is prime: True
115280095190773 is prime: True
115797848077099 is prime: True
1099726899285419 is prime: False
```

Another Example

In [57]:

```
from concurrent import futures
import urllib

URLS = ['http://localhost:8888/notebooks/PythonDesignPatterns.ipynb',
        'http://localhost:8888/static/base/images/logo.png',
        'http://localhost:8888/kernelspecs/python2/logo-64x64.png',
        'http://localhost:8888/tree',
        'http://localhost:8888/files/slides/ast_tree_elements.png']

def load_url(url):
    return urllib.urlopen(url).read()

def main():
    with futures.ThreadPoolExecutor(max_workers=5) as executor:
        future_to_url = dict((executor.submit(load_url, url), url)
                             for url in URLS)

        for future in futures.as_completed(future_to_url):
            url = future_to_url[future]
            try:
                print('%r page is %d bytes' % (url, len(future.result())))
            except Exception as e:
                print('%r generated an exception: %s' % (url, e))
```

In [56]:

```
main()
```

```
'http://localhost:8888/kernelspecs/python2/logo-64x64.png' page is 2180 bytes
'http://localhost:8888/notebooks/PythonDesignPatterns.ipynb' page is 22180 bytes
'http://localhost:8888/static/base/images/logo.png' page is 4473 bytes
'http://localhost:8888/files/slides/ast_tree_elements.png' page is 246982 bytes
'http://localhost:8888/tree' page is 12063 bytes
```

What we covered:

- Misunderstanding of Design Patterns
- General Idea of Python and Design Patterns
- **Creational** Design Patterns Practice

  - 4 GOF and 2 other patterns
- **Structural** Design Patterns Practice
  - 3 GOF, 2 FP and 1 other patterns
- **Behavioral** Design Patterns Practice
  - 5 GOF, 3 FP and 2 other patterns
- **Concurrent** Patterns
  - 6 current patterns

Now, You learned:

- Understand design pattern correctly, know when, how to use it
- Know most popular design pattern practices used in Python from GOF, Concurrency, Functional and Others popular ones.
- Know how to further learn design patterns and architectual pattern systematically


谢谢观看
wjo1212
wjo1212@163.com
LaiQiangDing
splunk>
listen to your data