

Ghi chú của một coder

Vũ Anh

Tháng 04 năm 2018

Mục lục

Mục lục	1
1 Nhập môn Python	2
1.0.1 Mục tiêu của khóa học	2
1.0.2 Đối tượng học viên	2
1.1 Giới thiệu	2
1.2 Cài đặt	3
1.2.1 Trên Windows	3
1.2.2 Trên CentOS	3
1.3 Biến - Hộp nhỏ	4
1.4 123s - Số trong Python	4
1.4.1 Print, print	4
1.4.2 Conditional	4
1.4.3 Loop	4
1.4.4 While Loop	4
1.4.5 For Loop	5
1.5 Cấu trúc dữ liệu	6
1.5.1 Number	6
1.5.2 Collection	7
1.5.3 String	9
1.5.4 Datetime	10
1.5.5 Object	11
1.6 Lập trình hướng đối tượng	11
1.6.1 Classes and Objects	11
1.6.2 self	12
1.6.3 Abstract Classes	14
1.6.4 Design Patterns	15
1.7 File System & IO	17
1.7.1 JSON	17
1.7.2 XML	18
2 Python ứng dụng	20
2.1 Yield and Generators	20
2.1.1 Metaclasses	24
2.2 Hệ điều hành	28
2.2.1 File Operations	28
2.2.2 CLI	28
2.3 Cơ sở dữ liệu (chưa xây dựng)	28

2.4	Giao diện (chưa xây dựng)	28
2.5	Lập trình mạng	28
3	Phát triển phần mềm với Python	29
3.0.1	Mục tiêu của khóa học	29
3.0.2	Đối tượng học viên	29
3.1	Logging	29
3.2	Configuration	30
3.3	Command Line	30
3.4	Testing	30
3.5	IDE & Debugging	31
3.5.1	Pycharm Pycharm	33
3.6	Package Manager	34
3.6.1	py2exe	34
3.6.2	Quản lý gói với Anaconda	34
3.7	Environment	35
3.7.1	Create	36
3.7.2	Clone	36
3.7.3	List	36
3.7.4	Remove	37
3.7.5	Share	37
3.8	Module	37
3.9	Production	39
3.10	Test với python	39
3.11	Xây dựng docs với readthedocs và sphinx	40

Tài liệu

43

Hướng dẫn online tại <http://magizbox.com/training/python/site/>

01/11/2017
Thích python
vì nó quá đơn
giản (và quá
đẹp).

Chương 1

Nhập môn Python

1.0.1 Mục tiêu của khóa học

Ưu điểm của khóa học

- Dành cho người mới bắt đầu, chưa từng học lập trình hoặc cho những ai muốn ôn lại kiến thức căn bản về lập trình python.
- Dễ học, dễ thực hành, ví dụ trực quan thú vị, không yêu cầu cao về máy móc hay phần mềm đi kèm.
- Ví dụ mẫu nhiều, trực quan, thú vị.

Kết thúc khóa học bạn sẽ học được gì?

- Xây dựng 5 dự án đơn giản với Python 3

Với những kiến thức bạn có thể làm gì?

- Lập trình viên Python tại các công ty phần mềm

1.0.2 Đối tượng học viên

- Những bạn chưa từng lập trình
- Những bạn đã có kinh nghiệm lập trình nhưng chưa lập trình python

1.1 Giới thiệu

Python is a general-purpose interpreted, interactive, object-oriented, and high-level programming language. It was created by Guido van Rossum during 1985-1990. Like Perl, Python source code is also available under the GNU General Public License (GPL). This tutorial gives enough understanding on Python programming language.

Python is Interpreted: Python is processed at runtime by the interpreter. You do not need to compile your program before executing it. This is similar to PERL and PHP.

Python is Interactive: You can actually sit at a Python prompt and interact with the interpreter directly to write your programs.

Python is Object-Oriented: Python supports Object-Oriented style or technique of programming that encapsulates code within objects.

Python is Beginner Friendly: Python is a great language for the beginner-level programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games.

Sách

[Tập hợp các sách python](#)

Khoá học

[Tập hợp các khóa học python](#)

Tham khảo

[Top 10 Python Libraries Of 2015](#)

1.2 Cài đặt

1.2.1 Trên Windows

Anaconda 4.3.0

Anaconda is BSD licensed which gives you permission to use Anaconda commercially and for redistribution.

1. Download the installer
 2. Optional: Verify data integrity with MD5 or SHA-256
 3. Double-click the .exe file to install Anaconda and follow the instructions on the screen
- Python 3.6 version 64-BIT INSTALLER Python 2.7 version 64-BIT INSTALLER
Step 2. Discover the Map
<https://docs.python.org/2/library/index.html>

1.2.2 Trên CentOS

Developer tools

The Development tools will allow you to build and compile software from source code. Tools for building RPMs are also included, as well as source code management tools like Git, SVN, and CVS.

```
yum groupinstall "Development tools"
yum install zlib-devel
yum install bzip2-devel
yum install openssl-devel
yum install ncurses-devel
yum install sqlite-devel
```

Python Anaconda Anaconda is BSD licensed which gives you permission to use Anaconda commercially and for redistribution.

```
cd /opt
wget --no-check-certificate https://www.python.org/ftp/python/2.7.6/
    ↪ Python-2.7.6.tar.xz
tar xf Python-2.7.6.tar.xz
cd Python-2.7.6
./configure --prefix=/usr/local
```

```
make && make altinstall
## link
ln -s /usr/local/bin/python2.7 /usr/local/bin/python
# final check
which python
python -V
# install Anaconda
cd ~/Downloads
wget https://repo.continuum.io/archive/Anaconda-2.3.0-Linux-x86_64.sh
bash ~/Downloads/Anaconda-2.3.0-Linux-x86_64.sh
```

1.3 Biến - Hộp nhỏ

1.4 123s - Số trong Python

1.4.1 Print, print

```
print "Hello World"
```

1.4.2 Conditional

```
if you_smart:
    print "learn python"
else:
    print "go away"
```

1.4.3 Loop

In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on. There may be a situation when you need to execute a block of code several number of times.

Programming languages provide various control structures that allow for more complicated execution paths. A loop statement allows us to execute a statement or group of statements multiple times. The following diagram illustrates a loop statement

Python programming language provides following types of loops to handle looping requirements.

while loop Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body. **for loop** Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable. **nested loops** You can use one or more loop inside any another while, for or do..while loop.

1.4.4 While Loop

A while loop statement in Python programming language repeatedly executes a target statement as long as a given condition is true.

Syntax

The syntax of a while loop in Python programming language is

```
while expression:
    statement(s)
```

Example

```
count = 0
while count < 9:
    print 'The count is:', count
    count += 1
print "Good bye!"
```

1.4.5 For Loop

It has the ability to iterate over the items of any sequence, such as a list or a string.

Syntax

```
for iterating_var in sequence:
    statements(s)
```

If a sequence contains an expression list, it is evaluated first. Then, the first item in the sequence is assigned to the iterating variable *iterating_var*. Next, the statements block is executed. Each iteration...

```
for i in range(10):
    print "hello", i

for letter in 'Python':
    print 'Current letter :', letter

fruits = ['banana', 'apple', 'mango']
for fruit in fruits:
    print 'Current fruit :', fruit

print "Good bye!"
```

Yield and Generator

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

```
def createGenerator():
    yield 1
    yield 2
    yield 3

mygenerator = createGenerator() # create a generator
print(mygenerator) # mygenerator is an object!
# <generator object createGenerator at 0xb7555c34>
for i in mygenerator:
    print(i)
# 1
# 2
# 3
```

Visit Yield and Generator explained for more information

Functions

Variable-length arguments

```
def functionname([formal_args,] *var_args_tuple ):
    "function_docstring"
    function_suite
    return [expression]
```

Example

```
#!/usr/bin/python

# Function definition is here
def printinfo( arg1, *vartuple ):
    "This prints a variable passed arguments"
    print "Output is: "
    print arg1
    for var in vartuple:
        print var
    return;

# Now you can call printinfo function
printinfo( 10 )
printinfo( 70, 60, 50 )
```

Coding Convention Code layout Indentation: 4 spaces

Suggest Readings

"Python Functions". www.tutorialspoint.com "Python Loops". www.tutorialspoint.com

"What does the “yield” keyword do?". stackoverflow.com "Improve Your Python:

'yield' and Generators Explained". jeffknupp.com

Vấn đề với mảng

Random Sampling ¹ - sinh ra một mảng ngẫu nhiên trong khoảng (0, 1), mảng ngẫu nhiên số nguyên trong khoảng (x, y), mảng ngẫu nhiên là permutation của số từ 1 đến n

1.5 Cấu trúc dữ liệu

1.5.1 Number

Basic Operation

```
1
1.2
1 + 2
abs(-5)
```

¹tham khảo [pytorch](<http://pytorch.org/docs/master/torch.html?highlight=randntorch.randn>), [numpy](<https://docs.scipy.org/doc/numpy-1.13.0/reference/routines.random.html>))

1.5.2 Collection

In this post I will cover 4 most popular data types in python list, tuple, set, dictionary

List The most basic data structure in Python is the sequence. Each element of a sequence is assigned a number - its position or index. The first index is zero, the second index is one, and so forth.

The list is a most versatile datatype available in Python which can be written as a list of comma-separated values (items) between square brackets. Important thing about a list is that items in a list need not be of the same type.

Usage

A list keeps order, dict and set don't: when you care about order, therefore, you must use list (if your choice of containers is limited to these three, of course)

Most Popular Operations

Create a list `a = ["a", "b", 3]` Access values in list `a[1]` Updated List `a[0] = 5` Delete list elements `del a[1]` Reverse a list `a[::-1]` Itertools `[a + b for (a, b) in itertools.product(x, y)]` Select random elements in list `random.choice(x)` `random.sample(x, 3)` Create a list `a = [1, 2, 3]` `[1, 2, 3]` Access values in list `list1 = ['physics', 'chemistry', 1997, 2000]` `list2 = [1, 2, 3, 4, 5, 6, 7]`

`print list1[0]` physics

`print list2[1:5]` [2, 3, 4, 5] Updated lists `list = ['physics', 'chemistry', 1997, 2000]`

`print list[2]` 1997

`list[2] = 2001` `print list[2]` 2001 Delete list elements `list1 = ['physics', 'chemistry', 1997, 2000];`

`print list1` ['physics', 'chemistry', 1997, 2000]

`del list1[2]`

`print list1` ['physics', 'chemistry', 2000] Reverse a list `[1, 3, 2][::-1]` [2, 3, 1]

Itertools `import itertools`

`x = [1, 2, 3]` `y = [2, 4, 5]`

`[a + b for (a, b) in itertools.product(x, y)]` [3, 5, 6, 4, 6, 7, 5, 7, 8] Select random elements in list `import random`

`x = [13, 23, 14, 52, 6, 23]`

`random.choice(x)` 52

`random.sample(x, 3)` [23, 14, 52] **Tuples** A tuple is a sequence of immutable Python objects. Tuples are sequences, just like lists. The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use parentheses, whereas lists use square brackets.

Usage

Tuples have structure, lists have order Tuples being immutable there is also a semantic distinction that should guide their usage. Tuples are heterogeneous data structures (i.e., their entries have different meanings), while lists are homogeneous sequences **Most Popular Operations**

Create a tuple `t = ("a", 1, 2)` Accessing Values in Tuples `t[0]`, `t[1:]` Updating Tuples Not allowed Create a tuple `tup1 = ('physics', 'chemistry', 1997, 2000);`

`tup2 = (1, 2, 3, 4, 5);` `tup3 = "a", "b", "c", "d";` `tup4 = ()` `tup5 = (50,)`

Accessing Values in Tuples `!usr/bin/python`

`tup1 = ('physics', 'chemistry', 1997, 2000);` `tup2 = (1, 2, 3, 4, 5, 6, 7);`

`tup1[0]` physics

`tup2[1:5]` [2, 3, 4, 5] **Updating Tuples** Tuples are immutable which means you cannot update or change the values of tuple elements. You are able to take por-

tions of existing tuples to create new tuples as the following example demonstrates

```
tup1 = (12, 34.56); tup2 = ('abc', 'xyz');
```

Following action is not valid for tuples `tup1[0] = 100;`

So let's create a new tuple as follows `tup3 = tup1 + tup2;` print `tup3` Set Sets are lists with no duplicate entries.

The sets module provides classes for constructing and manipulating unordered collections of unique elements. Common uses include membership testing, removing duplicates from a sequence, and computing standard math operations on sets such as intersection, union, difference, and symmetric difference.

Usage

set forbids duplicates, list does not: also a crucial distinction. Most Popular Operations

Create a set `x = set(["Postcard", "Radio", "Telegram"])` Add elements to a set `x.add("Mobile")` Remove elements to a set `x.remove("Radio")` Subset `y.issubset(x)` Intersection `x.intersection(y)` Difference between two sets `x.difference(y)` Create a set `x = set(["Postcard", "Radio", "Telegram"])` `x = set(['Postcard', 'Telegram', 'Radio'])` Add elements to a set `x = set(["Postcard", "Radio", "Telegram"])` `x.add("Mobile")` `x = set(['Postcard', 'Telegram', 'Mobile', 'Radio'])` Remove elements to a set `x = set(["Postcard", "Radio", "Telegram"])` `x.remove("Radio")` `x = set(['Postcard', 'Telegram'])` Subset `x = set(["a", "b", "c", "d"])` `y = set(["c", "d"])` `y.issubset(x)` True Intersection `x = set(["a", "b", "c", "d"])` `y = set(["c", "d"])` `x.intersection(y)` `set(['c', 'd'])` Difference between two sets `x = set(["Postcard", "Radio", "Telegram"])` `y = set(["Radio", "Television"])` `x.difference(y)` `set(['Postcard', 'Telegram'])` Dictionary Each key is separated from its value by a colon (:), the items are separated by commas, and the whole thing is enclosed in curly braces. An empty dictionary without any items is written with just two curly braces, like this: .

Keys are unique within a dictionary while values may not be. The values of a dictionary can be of any type, but the keys must be of an immutable data type such as strings, numbers, or tuples.

Usage

dict associates with each key a value, while list and set just contain values: very different use cases, obviously. Most Popular Operations

Create a dictionary `d = {"a": 1, "b": 2, "c": 3}` Update dictionary `d["a"] = 4`

Delete dictionary elements `del d["a"]` Create a dictionary `dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}`

`print "dict['Name']:", dict['Name']` `print "dict['Age']:", dict['Age']` Update dictionary `dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}`

`dict['Age'] = 8;` update existing entry `dict['School'] = "DPS School";` Add new entry

`print "dict['Age']:", dict['Age']` `print "dict['School']:", dict['School']` Delete dictionary elements `dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}`

`del dict['Name'];` remove entry with key 'Name' `dict.clear();` remove all entries in dict `del dict ;` delete entire dictionary

`print "dict['Age']:", dict['Age']` `print "dict['School']:", dict['School']` Related Readings Python Lists, tutorialspoint.com Python Dictionary, tutorialspoint.com Python Dictionary Methods, [guru99](http://guru99.com) In Python, when to use a Dictionary, List or Set?, [stackoverflow](http://stackoverflow.com) What's the difference between lists and tuples?, [stackoverflow](http://stackoverflow.com)

1.5.3 String

Format '0, 1, 2'.format('a', 'b', 'c') 'a, b, c' Regular Expressions The aim of this chapter of our Python tutorial is to present a detailed led and descriptive introduction into regular expressions. This introduction will explain the theoretical aspects of regular expressions and will show you how to use them in Python scripts.

Regular Expressions are used in programming languages to filter texts or textstrings.

It's possible to check, if a text or a string matches a regular expression.

There is an aspect of regular expressions which shouldn't go unmentioned: The syntax of regular expressions is the same for all programming and script languages, e.g. Python, Perl, Java, SED, AWK and even X.

Functions match function This function attempts to match RE pattern to string with optional flags.

re.match(pattern, string, flags=0) Example

```
import re
```

```
line = "Cats are smarter than dogs"
```

```
matched_object = re.match(r'(.*)are(.?)*.*', line, re.M|re.I)
```

```
if matched_object : print"matched_object.group() : ", matched_object.group()print"matched_object.group(1) :
```

```
", matched_object.group(1)print"matched_object.group(2) : ", matched_object.group(2)else :
```

```
print"Nomatch!!" Whenthecodeisexecuted, itproducesfollowingresults
```

```
matched_object.group() : Catsaresmarterthandogsmatched_object.group(1) : Catsmatched_object.group(2) :
```

```
smartersearchfunctionThisfunctionsearchesforfirstoccurrenceofREpatternwithinstirngwithoptional f
```

```
re.search(pattern, string, flags=0) Example
```

```
!/usr/bin/python import re
```

```
line = "Cats are smarter than dogs"
```

```
search_object = re.search(r'dogs', line, re.M|re.I)if search_object : print"search—
```

```
— > search_object.group() : ", search_object.group()else : print"Nothingfound!!" Whenthecodeisexecuted, itp
```

```
search—> search_object.group() : dogssubfunctionThismethodreplacesalloccurrencesoftheREpatterninstring
```

```
re.sub(pattern, repl, string, max=0) Example
```

```
!/usr/bin/python import re
```

```
phone = "2004-959-559 This is Phone Number"
```

```
Delete Python-style comments num = re.sub(r'.*', "", phone)print"PhoneNum :
```

```
", num
```

```
Remove anything other than digits num = re.sub(r", """, phone) print "Phone
```

```
Num : ", num When the code is executed, it produces following results
```

```
Phone Num : 2004-959-559 Phone Num : 2004959559 Tokens Cheatsheet Char-
```

```
acter Classes . any character except newline /go.gle/ google goggle gogle word,
```

```
digit, whitespace // AaYyz09 ?! // 012345 aZ? // 0123456789 abcd?/ $not word,
```

```
digit, whitespace // abcded 1234 ?> // abc 12345 ?<. /$/ abc 123? <. [abc] any
```

```
of a, b or c /analy[sz]e/ analyse analyze analyxe [abc]nota, borc/analy[sz]e/analyseanalyzeanalyxe[a—
```

```
g]characterbetweenag/[2—4]/demo1demo2demo3demo4demo5QuantifiersAlternation*
```

```
a+a?0ormore, 1ormore, 0or1/go*gle/goglegoglegooglegooooooglehgle/go+gle/gglegoglegooglegooooooglehgle,
```

```
start / end of the string /abc/ abc /abc/abcabc/abc/ abc abc_word, not-word
```

```
boundary // This island is beautiful. // cat certificate Escaped characters
```

```
escaped special characters // username@exampe.com 300.000 USD // abc@/
```

```
/ abc@ tab, linefeed, carriage return // abc def /ab/ ab // abc@00A9 unicode es-
```

```
caped © /00A9/ Copyright©2017 - All rights reserved Groups and Lockaround
```

```
(abc) capture group /(demo|example)[0-9]/ demo1example4demo backreference
```

to group 1 `/(abc|def)=/` `abc=abc def=defabc=def` `(?:abc)` non-capturing group
`/(?:abc)3/` `abccabccabcc` `abcabc` `(?=abc)` positive lookahead `/t(?:=s)/` `tttssstttss`
`(?!abc)` negative lookahead `/t(?:!s)/` `tttssstttss` `(?<=abc)` positive lookbehind
`/(?<=foo)bar/` `foobar fuubar` `(?<!abc)` negative lookbehind `/(?<!foo)bar/` `foo-`
`bar fuubar` Related Readings

Online regex tester and debugger: PHP, PCRE, Python, Golang and JavaScript,
regex101.com RegExr: Learn, Build, Test RegEx, regexr.com

1.5.4 Datetime

Print current time

```
from datetime import datetime
datetime.now().strftime(' %Y-%m-%d %H:%M:%S')
```

Get current time

```
import datetime
datetime.datetime.now() datetime(2009, 1, 6, 15, 8, 24, 78915)
```

Unixtime

```
import time
int(time.time()) Measure time elapsed
```

import time

```
start = time.time()
print("hello")
end = time.time()
print(end - start) Moment
```

Dealing with dates in Python shouldn't have to suck.

Installation

```
pip install moment Usage
```

```
import moment
from datetime import datetime
```

```
Create a moment from a string
moment.date("12-18-2012")
```

```
Create a moment with a specified strftime format
moment.date("12-18-2012", "%Y-%m-%d")
```

```
Moment uses the awesome dateparser library behind the scenes
moment.date("2012-12-18")
```

```
Create a moment with words in it
moment.date("December 18, 2012")
```

```
Create a moment that would normally be pretty hard to do
moment.date("2 weeks ago")
```

```
Create a future moment that would otherwise be really difficult
moment.date("2 weeks from now")
```

```
Create a moment from the current datetime
moment.now()
```

```
The moment can also be UTC-based
moment.utcnow()
```

```
Create a moment with the UTC time zone
moment.utc("2012-12-18")
```

```
Create a moment from a Unix timestamp
moment.unix(1355875153626)
```

```
Create a moment from a Unix UTC timestamp
moment.unix(1355875153626, utc=True)
```

```
Return a datetime instance
moment.date(2012, 12, 18).date
```

```
We can do the same thing with the UTC method
moment.utc(2012, 12, 18).date
```

```
Create and format a moment using Moment.js semantics
moment.now().format("YYYY-MM-DD")
```

```
Create and format a moment with strftime semantics
moment.date(2012, 12, 18).strftime("%Y-%m-%d")
```

```
Update your moment's time zone
moment.date(datetime(2012, 12, 18)).locale("US/Central").date
```

```
Alter the moment's UTC time zone to a different time zone
moment.utcnow().timezone("US/Eastern").date
```

```
Set and update your moment's time zone. For instance, I'm on the west coast,
but want NYC's current time.
moment.now().locale("US/Pacific").timezone("US/Eastern")
```

```
In order to manipulate time zones, a locale must always be set or you must be
using UTC.
moment.utcnow().timezone("US/Eastern").date
```

You can also clone a moment, so the original stays unaltered `now = moment.utcnow().timezone("US/Pacific")` `future = now.clone().add(weeks=2)` Related Readings How to get current time in Python, [stackoverflow](#) Does Python's `time.time()` return the local or UTC timestamp?, [stackoverflow](#) Measure time elapsed in Python?, [stackoverflow](#) [momnet, https://github.com/zachwill/moment](https://github.com/zachwill/moment)

1.5.5 Object

Convert dict to object Elegant way to convert a normal Python dict with some nested dicts to an object

```
class Struct:
    def __init__(self, **entries):
        self.__dict__.update(entries)
    Then, you can use
> args = 'a': 1, 'b': 2
> s = Struct(**args)
> s < main.Struct instance at 0x01D6A738 >
> s.a1 > s.b2
Related Readings
stackoverflow, Convert Python dict to object?
```

1.6 Lập trình hướng đối tượng

Object Oriented Programming Python has been an object-oriented language since it existed. Because of this, creating and using classes and objects are downright easy. This chapter helps you become an expert in using Python's object-oriented programming support.

If you do not have any previous experience with object-oriented (OO) programming, you may want to consult an introductory course on it or at least a tutorial of some sort so that you have a grasp of the basic concepts.

1.6.1 Classes and Objects

Classes can be thought of as blueprints for creating objects. When I define a `BankAccount` class using the `class` keyword, I haven't actually created a bank account. Instead, what I've created is a sort of instruction manual for constructing "bank account" objects. Let's look at the following example code:

```
class BankAccount:
    id = None
    balance = 0

    def __init__(self, id, balance=0):
        self.id = id
        self.balance = balance

    def __get_balance(self):
        return self.balance

    def withdraw(self, amount):
        self.balance = self.balance - amount

    def deposit(self, amount):
        self.balance = self.balance + amount

john = BankAccount(1, 1000.0)
john.withdraw(100.0)
```

The class `BankAccount` line does not create a new bank account. That is, just because we've defined a `BankAccount` doesn't mean we've created one; we've merely outlined the blueprint to create a `BankAccount` object. To do so, we call the class's *`__init__` method with the proper number of arguments (minus `self`, which we'll get to in a moment)*. So, to use the "blueprint" that we created by defining the class `BankAccount` (which is used to create `BankAccount` objects), we call the class name almost as if it were a function: `john = BankAccount(1, 1000.0)`. This line simply says "use the `BankAccount` blueprint to create me a new object, which I'll refer to as `john`".

The `john` object, known as an instance, is the realized version of the `BankAccount` class. Before we called `BankAccount()`, no `BankAccount` object existed. We can, of course, create as many `BankAccount` objects as we'd like. There is still, however, only one `BankAccount` class, regardless of how many instances of the class we create.

1.6.2 self

So what's with that `self` parameter to all of the `BankAccount` methods? What is it? Why, it's the instance, of course! Put another way, a method like `withdraw` defines the instructions for withdrawing money from some abstract customer's account. Calling `john.withdraw(100)` puts those instructions to use on the `john` instance.

So when we say `def withdraw(self, amount):`, we're saying, "here's how you withdraw money from a `BankAccount` object (which we'll call `self`) and a dollar figure (which we'll call `amount`). `self` is the instance of the `BankAccount` that `withdraw` is being called on. That's not me making analogies, either. `john.withdraw(100.0)` is just shorthand for `BankAccount.withdraw(john, 100.0)`, which is perfectly valid (if not often seen) code.

Constructors: *`__init__`*

`self` may make sense for other methods, but what about *`__init__`? When we call `__init__`, we're in the process of creating an object, so how can the*

This is why when we call *`__init__`, we initialize objects by saying things like `self.id = id`. Remember, since `self` is the instance, this is equivalent to saying*
Be careful what you *`__init__`*

After *`__init__` has finished, the caller can rightly assume that the object is ready to use. That is, after `john = BankAccount(1, 1000.0)`, we can start making deposits*

Inheritance While Object-oriented Programming is useful as a modeling tool, it truly gains power when the concept of inheritance is introduced. Inheritance is the process by which a "child" class derives the data and behavior of a "parent" class. An example will definitely help us here.

Imagine we run a car dealership. We sell all types of vehicles, from motorcycles to trucks. We set ourselves apart from the competition by our prices. Specifically, how we determine the price of a vehicle on our lot: \$5,000 x number of wheels a vehicle has. We love buying back our vehicles as well. We offer a flat rate - 10% of the miles driven on the vehicle. For trucks, that rate is \$10,000. For cars, \$8,000. For motorcycles, \$4,000.

If we wanted to create a sales system for our dealership using Object-oriented techniques, how would we do so? What would the objects be? We might have a `Sale` class, a `Customer` class, an `Inventory` class, and so forth, but we'd almost certainly have a `Car`, `Truck`, and `Motorcycle` class.

What would these classes look like? Using what we've learned, here's a possible implementation of the `Car` class:

```

class Car(object):
    def __init__(self, wheels, miles, make, model, year, sold_on):
        self.wheels = wheels
        self.miles = miles
        self.make = make
        self.model = model
        self.year = year
        self.sold_on = sold_on

    def sale_price(self):
        if self.sold_on is not None:
            return 0.0 # Already sold
        return 5000.0 * self.wheels

    def purchase_price(self):
        if self.sold_on is None:
            return 0.0 # Not yet sold
        return 8000 - (.10 * self.miles)

```

OK, that looks pretty reasonable. Of course, we would likely have a number of other methods on the class, but I've shown two of particular interest to us: *sale_price* and *purchase_price*. *We'll see why these are important in a bit.*

Now that we've got the Car class, perhaps we should create a Truck class? Let's follow the same pattern we did for car:

```

class Truck(object):
    def __init__(self, wheels, miles, make, model, year, sold_on):
        self.wheels = wheels
        self.miles = miles
        self.make = make
        self.model = model
        self.year = year
        self.sold_on = sold_on

    def sale_price(self):
        if self.sold_on is not None:
            return 0.0 # Already sold
        return 5000.0 * self.wheels

    def purchase_price(self):
        if self.sold_on is None:
            return 0.0 # Not yet sold
        return 10000 - (.10 * self.miles)

```

Wow. That's almost identical to the car class. One of the most important rules of programming (in general, not just when dealing with objects) is "DRY" or "Don't Repeat Yourself. We've definitely repeated ourselves here. In fact, the Car and Truck classes differ only by a single character (aside from comments). So what gives? Where did we go wrong? Our main problem is that we raced straight to the concrete: Car and Truck are real things, tangible objects that make intuitive sense as classes. However, they share so much data and function-

ality in common that it seems there must be an abstraction we can introduce here. Indeed there is: the notion of Vehicle.

1.6.3 Abstract Classes

A Vehicle is not a real-world object. Rather, it is a concept that some real-world objects (like cars, trucks, and motorcycles) embody. We would like to use the fact that each of these objects can be considered a vehicle to remove repeated code. We can do that by creating a Vehicle class:

```
class Vehicle(object):
    base_sale_price = 0

    def __init__(self, wheels, miles, make, model, year, sold_on):
        self.wheels = wheels
        self.miles = miles
        self.make = make
        self.model = model
        self.year = year
        self.sold_on = sold_on

    def sale_price(self):
        if self.sold_on is not None:
            return 0.0 # Already sold
        return 5000.0 * self.wheels

    def purchase_price(self):
        if self.sold_on is None:
            return 0.0 # Not yet sold
        return self.base_sale_price - (.10 * self.miles)
```

Now we can make the Car and Truck class inherit from the Vehicle class by replacing object in the line class Car(object). The class in parenthesis is the class that is inherited from (object essentially means "no inheritance". We'll discuss exactly why we write that in a bit).

We can now define Car and Truck in a very straightforward way:

```
class Car(Vehicle):

    def __init__(self, wheels, miles, make, model, year, sold_on):
        self.wheels = wheels
        self.miles = miles
        self.make = make
        self.model = model
        self.year = year
        self.sold_on = sold_on
        self.base_sale_price = 8000

class Truck(Vehicle):
```



```
def __init__(self, wheels, miles, make, model, year, sold_on):
    self.wheels = wheels
    self.miles = miles
    self.make = make
    self.model = model
    self.year = year
    self.sold_on = sold_on
    self.base_sale_price = 10000
```

Object Convert dict to object

```
class Struct:
    def __init__(self, **entries):
        self.__dict__.update(entries)
```

Then, you can use

```
> args = {'a': 1, 'b': 2}
> s = Struct(**args)
> s
< __main__.Struct instance at 0x01D6A738 >
> s.a
1
> s.b
2
```

Suggested Readings Improve Your Python: Python Classes and Object Oriented Programming stackoverflow, Convert Python dict to object? Why are Python's 'private' methods not actually private?

1.6.4 Design Patterns

Design Patterns Singleton Non-thread-safe Paul Manta's implementation of singletons

```
@Singleton
class Foo:
    def __init__(self):
        print 'Foo created'

f = Foo() # Error, this isn't how you get the instance of a singleton

f = Foo.Instance() # Good. Being explicit is in line with the Python Zen
g = Foo.Instance() # Returns already created instance

print f is g # True

class Singleton:
    """
    A non-thread-safe helper class to ease implementing singletons.
    This should be used as a decorator -- not a metaclass -- to the
```

class that should be a singleton.

The decorated class can define one `__init__` function that takes only the `self` argument. Also, the decorated class cannot be inherited from. Other than that, there are no restrictions that apply to the decorated class.

To get the singleton instance, use the `Instance` method. Trying to use `__call__` will result in a `TypeError` being raised.

```
"""
def __init__(self, decorated):
    self._decorated = decorated

def Instance(self):
    """
    Returns the singleton instance. Upon its first call, it creates a
    new instance of the decorated class and calls its __init__
    ↪ method.
    On all subsequent calls, the already created instance is returned.
    """
    try:
        return self._instance
    except AttributeError:
        self._instance = self._decorated()
        return self._instance

def __call__(self):
    raise TypeError('Singletons must be accessed through Instance()'.
    ↪ ')

def __instancecheck__(self, inst):
    return isinstance(inst, self._decorated)
```

Thread safe

werediver's implementation of singletons. A thread safe implementation of singleton pattern in Python. Based on `tornado.ioloop.IOLoop.instance()` approach.

import threading

Based on `tornado.ioloop.IOLoop.instance()` approach. See <https://github.com/facebook/tornado>

```
class SingletonMixin(object):
    _singleton_lock = threading.Lock()
    _singleton_instance = None
```

```
@classmethod
def instance(cls):
    if not cls._singleton_instance:
        with cls._singleton_lock:
            if not cls._singleton_instance:
                cls._singleton_instance = cls()
```

```
class A(SingletonMixin):
    pass
```

```
class B(SingletonMixin):
    pass
```

```
if __name__ == '__main__':
    a, a2 = A.instance(), A.instance()
    b, b2 = B.instance(), B.instance()
```

```
assert a is a2
assert b is b2
assert a is not b
```

```
print('a: ', a)
print('b: ', b)
Suggested Readings
Is there a simple, elegant way to define singletons?
```

1.7 File System & IO

1.7.1 JSON

Write json file with pretty format and unicode

```
import json
import io

data = {
    "menu": {
        "header": "Sample Menu",
        "items": [
            {"id": "Open"},
            {"id": "OpenNew", "label": "Open New"},
            None,
            {"id": "Help"},
            {"id": "About", "label": "About Adobe CVG Viewer..."}
        ]
    }
}

with io.open("sample_json.json", "w", encoding="utf8") as f:
    content = json.dumps(data, indent=4, sort_keys=True, ensure_ascii=
        ↪ False)
    f.write(unicode(content))
```

Output

```
{
  "menu": {
    "header": "Sample Menu",
    "items": [
      {
        "id": "Open"
      },
      {
        "id": "OpenNew",
        "label": "Open New"
      },
      null,
      {
        "id": "Help"
      },
      {
        "id": "About",
        "label": "About Adobe CVG Viewer..."
      }
    ]
  }
}
```

Read json file

```
import json
from pprint import pprint

with open('sample_json.json') as data_file:
    data = json.load(data_file)

pprint(data)
```

Output

```
{u'menu': {u'header': u'Sample Menu',
           u'items': [{u'id': u'Open'},
                      {u'id': u'OpenNew', u'label': u'Open New'},
                      None,
                      {u'id': u'Help'},
                      {u'id': u'About',
                       u'label': u'About Adobe CVG Viewer...'}]}}
```

Related Reading

Parsing values from a JSON file in Python, [stackoverflow](#) How do I write JSON data to a file in Python?, [stackoverflow](#)

1.7.2 XML

Write xml file with lxml package

```
import lxml.etree as ET
# root declaration
root = ET.Element('catalog')
# insert comment
comment = ET.Comment(' this is a xml sample file ')
root.insert(1, comment)
# book element
book = ET.SubElement(root, 'book', id="bk001")
# book data
author = ET.SubElement(book, 'author')
author.text = "Gambardella, Matthew"
title = ET.SubElement(book, 'title')
title.text = "XML Developer's Guide"
# write xml to file
tree = ET.ElementTree(root)
tree.write("sample_book.xml", pretty_print=True, xml_declaration=
    ↪ True, encoding='utf-8')
```

Output

```
<?xml version='1.0' encoding='UTF-8'?>
<catalog>
  <!-- this is a xml sample file -->
  <book id="bk001">
    <author>Gambardella, Matthew</author>
    <title>XML Developer's Guide</title>
  </book>
```

```
</catalog>
```

Read xml file with lxml package

```
from lxml import etree as ET

tree = ET.parse("sample_book.xml")
root = tree.getroot()
book = root.find('book')
print "Book Information"
print "ID      :", book.attrib["id"]
print "Author :", book.find('author').text
print "Title  :", book.find('title').text
```

Output

```
Book Information
ID      : bk001
Author : Gambardella, Matthew
Title  : XML Developer's Guide
```

Chương 2

Python ứng dụng

Mục tiêu của khoá học

Tìm hiểu các vấn đề lập trình Python nâng cao qua các ví dụ thực tế, sinh động

Đối tượng học viên

- Là sinh viên năm 2, năm 3
- Đang học các môn Lập trình song song, phát triển Web

2.1 Yield and Generators

Coroutines and Subroutines When we call a normal Python function, execution starts at function's first line and continues until a return statement, exception, or the end of the function (which is seen as an implicit return None) is encountered. Once a function returns control to its caller, that's it. Any work done by the function and stored in local variables is lost. A new call to the function creates everything from scratch.

This is all very standard when discussing functions (more generally referred to as subroutines) in computer programming. There are times, though, when it's beneficial to have the ability to create a "function" which, instead of simply returning a single value, is able to yield a series of values. To do so, such a function would need to be able to "save its work," so to speak.

I said, "yield a series of values" because our hypothetical function doesn't "return" in the normal sense. return implies that the function is returning control of execution to the point where the function was called. "Yield," however, implies that the transfer of control is temporary and voluntary, and our function expects to regain it in the future.

In Python, "functions" with these capabilities are called generators, and they're incredibly useful. generators (and the yield statement) were initially introduced to give programmers a more straightforward way to write code responsible for producing a series of values. Previously, creating something like a random number generator required a class or module that both generated values and kept track of state between calls. With the introduction of generators, this became much simpler.

To better understand the problem generators solve, let's take a look at an example. Throughout the example, keep in mind the core problem being solved: generating a series of values.

Note: Outside of Python, all but the simplest generators would be referred to as coroutines. I'll use the latter term later in the post. The important thing to remember is, in Python, everything described here as a coroutine is still a generator. Python formally defines the term generator; coroutine is used in discussion but has no formal definition in the language.

Example: Fun With Prime Numbers Suppose our boss asks us to write a function that takes a list of ints and returns some Iterable containing the elements which are prime numbers.

Remember, an Iterable is just an object capable of returning its members one at a time.

"Simple," we say, and we write the following:

```
def get_primes(input_list):
    result_list = list()
    for element in input_list:
        if is_prime(element):
            result_list.append()

    return result_list
```

or better yet...

```
def get_primes(input_list):
    return (element for element in input_list if is_prime(element))

# not germane to the example, but here's a possible implementation of
# is_prime...

def is_prime(number):
    if number > 1:
        if number == 2:
            return True
        if number % 2 == 0:
            return False
        for current in range(3, int(math.sqrt(number) + 1), 2):
            if number % current == 0:
                return False
        return True
    return False
```

Either `get_primes` implementation above fulfills the requirements, so we tell our boss we're done. She reports our function is working well, but not quite exactly. A few days later, our boss comes back and tells us she's run into a small problem: she wants to use our `get_primes` function on a very large list of numbers. In fact, the list is so large that merely creating it would consume too much memory. Once we think about this new requirement, it becomes clear that it requires more than a simple change to `get_primes`. Clearly, we can't return a list of all the prime numbers from start to infinity (or even a very large number). Before we give up, let's determine the core obstacle preventing us from writing a function that satisfies our boss's new requirements. Thinking about it, we arrive at the following: functions only get one chance to return results, and thus must

return all results at once. It seems pointless to make such an obvious statement; "functions just work that way," we think. The real value lies in asking, "but what if they didn't?"

Imagine what we could do if *get_pprimes* could simply return the next value instead of all the values at once. It would be great. Unfortunately, this doesn't seem possible. Even if we had a magical function that allowed us to iterate from *n* to infinity, we'd get stuck after returning the first value:

```
def getpprimes(start) : for element in magical_infinite_range(start) : if ispprime(element) :
return element
Imagine getpprimes is called like so :
def solvenumber10() : She * is * working on Project Euler 10, I know it! total =
2 for nextpprime in getpprimes(3) : if nextpprime < 2000000 : total += nextpprime else :
print(total) return
Clearly, in getpprimes, we would immediately hit the case where number =
3 and return at line 4. Instead of return, we need a way to generate a value and, when asked for the next one, pick up
```

Functions, though, can't do this. When they return, they're done for good. Even if we could guarantee a function would be called again, we have no way of saying, "OK, now, instead of starting at the first line like we normally do, start up where we left off at line 4." Functions have a single entry point: the first line.

Enter the Generator This sort of problem is so common that a new construct was added to Python to solve it: the generator. A generator "generates" values. Creating generators was made as straightforward as possible through the concept of generator functions, introduced simultaneously.

A generator function is defined like a normal function, but whenever it needs to generate a value, it does so with the `yield` keyword rather than `return`. If the body of a `def` contains `yield`, the function automatically becomes a generator function (even if it also contains a `return` statement). There's nothing else we need to do to create one.

generator functions create generator iterators. That's the last time you'll see the term generator iterator, though, since they're almost always referred to as "generators". Just remember that a generator is a special type of iterator. To be considered an iterator, generators must define a few methods, one of which is `next()`. To get the next value from a generator, we use the same built-in function as for iterators: `next()`.

This point bears repeating: to get the next value from a generator, we use the same built-in function as for iterators: `next()`.

(`next()` takes care of calling the generator's `next()` method). Since a generator is a type of iterator, it can be used in a `for` loop.

So whenever `next()` is called on a generator, the generator is responsible for passing back a value to whomever called `next()`. It does so by calling `yield` along with the value to be passed back (e.g. `yield 7`). The easiest way to remember what `yield` does is to think of it as `return` (plus a little magic) for generator functions.**

Again, this bears repeating: `yield` is just `return` (plus a little magic) for generator functions.

Here's a simple generator function:

```
>>> def simple_generator_function() :>>> yield 1 >>> yield 2 >>> yield 3
And here are two simple ways to use it
>>> for value in simple_generator_function() :>>> print(value)
123 >>> our_generator = simple_generator_function() >>> next(our_generator)
1 >>> next(our_generator)
2 >>> next(our_generator)
3
Magic? What's the magic part? Glad you asked! When a generator function calls yield, the "return" value is the value after yield.
Let's rewrite getpprimes as a generator function. Notice that we no longer need the magical_infinite_range function
```



```

import random

def get_data():
    """Return 3 random integers between 0 and 9"""
    return random.sample(range(10), 3)

def consume():
    """Displays a running average across lists of integers sent to it"""
    running_sum = 0
    data_items_seen = 0

    while True:
        data = yield
        data_items_seen += len(data)
        running_sum += sum(data)
        print('The running average is {}'.format(running_sum / float(
            ↪ data_items_seen)))

def produce(consumer):
    """Produces a set of values and forwards them to the pre-defined
    ↪ consumer
    function"""
    while True:
        data = get_data()
        print('Produced {}'.format(data))
        consumer.send(data)
        yield

if __name__ == '__main__':
    consumer = consume()
    consumer.send(None)
    producer = produce(consumer)

    for _ in range(10):
        print('Producing...')
        next(producer)

```

Remember... There are a few key ideas I hope you take away from this discussion: generators are used to generate a series of values yield is like the return of generator functions The only other thing yield does is save the "state" of a generator function A generator is just a special type of iterator Like iterators, we can get the next value from a generator using next() for gets values by calling next() implicitly

2.1.1 Metaclasses

Metaclasses Python, Classes, and Objects Most readers are aware that Python is an object-oriented language. By object-oriented, we mean that Python can define classes, which bundle data and functionality into one entity. For example, we may create a class IntContainer which stores an integer and allows certain

operations to be performed:

```
class IntContainer(object):
    def __init__(self, i):
        self.i = int(i)

    def add_one(self):
        self.i += 1
ic = IntContainer(2)
ic.add_one()
print(ic.i)
3
```

This is a bit of a silly example, but shows the fundamental nature of classes: their ability to bundle data and operations into a single object, which leads to cleaner, more manageable, and more adaptable code. Additionally, classes can inherit properties from parents and add or specialize attributes and methods. This object-oriented approach to programming can be very intuitive and powerful. What many do not realize, though, is that quite literally everything in the Python language is an object.

For example, integers are simply instances of the built-in `int` type:

`print type(1) <type 'int'>` To emphasize that the `int` type really is an object, let's derive from it and specialize the *`add` method (which is the machinery underneath the `+` operator)*: (Note: We'll use the super syntax to call methods from the parent class: if you're unfamiliar with this, take a look at this [StackOverflow question](#)).

```
class MyInt(int):
    def __add__(self, other):
        print "specializing addition"
        return super(MyInt, self).__add__(other)

i = MyInt(2)
print(i + 2)
specializing addition
4
```

Using the `+` operator on our derived type goes through our *`add` method, as expected. We see that `int` really is an object that can be subclassed*.
Down the Rabbit Hole: Classes as Objects We said above that everything in python is an object: it turns out that this is true of classes themselves. Let's look at an example.

We'll start by defining a class that does nothing

`class DoNothing(object):` pass If we instantiate this, we can use the type operator to see the type of object that it is:

`d = DoNothing()` `type(d)` *`main.DoNothing` We see that our variable is an instance of the class `main.DoNothing`.*

We can do this similarly for built-in types:

`L = [1, 2, 3]` `type(L)` `list` A list is, as you may expect, an object of type `list`.

But let's take this a step further: what is the type of `DoNothing` itself?

`type(DoNothing)` `type` The type of `DoNothing` is `type`. This tells us that the class `DoNothing` is itself an object, and that object is of type `type`.

It turns out that this is the same for built-in datatypes:

`type(tuple)`, `type(list)`, `type(int)`, `type(float)` (`type`, `type`, `type`, `type`) What this shows is that in Python, classes are objects, and they are objects of type `type`.

type is a metaclass: a class which instantiates classes. All new-style classes in Python are instances of the type metaclass, including type itself:

type(type) type Yes, you read that correctly: the type of type is type. In other words, type is an instance of itself. This sort of circularity cannot (to my knowledge) be duplicated in pure Python, and the behavior is created through a bit of a hack at the implementation level of Python.

Metaprogramming: Creating Classes on the Fly Now that we've stepped back and considered the fact that classes in Python are simply objects like everything else, we can think about what is known as metaprogramming. You're probably used to creating functions which return objects. We can think of these functions as an object factory: they take some arguments, create an object, and return it. Here is a simple example of a function which creates an int object:

```
def int_factory(s) : i = int(s) return i
```

i = int_factory('100') print(i) 100 *This is overly-simplistic, but any function you write in the course of a normal program takes some arguments, does some operations, and creates and returns an object. With the above discussion in mind, though, this is a metafunction :*

```
def class_factory(object) : pass return Foo
```

F = class_factory(f = F()) print(type(f)) <class 'main.Foo'> *Just as the function int_factory constructs and returns an instance of int,*

But the above construction is a bit awkward: especially if we were going to do some more complicated logic when constructing Foo, it would be nice to avoid all the nested indentations and define the class in a more dynamic way. We can accomplish this by instantiating Foo from type directly:

```
def class_factory() : return type('Foo', (), )
```

F = class_factory() f = F() print(type(f)) <class 'main.Foo'> *In fact, the construct*

class MyClass(object): pass is identical to the construct

MyClass = type('MyClass', (),) MyClass is an instance of type type, and that can be seen explicitly in the second version of the definition. A potential confusion arises from the more common use of type as a function to determine the type of an object, but you should strive to separate these two uses of the keyword in your mind: here type is a class (more precisely, a metaclass), and MyClass is an instance of type.

The arguments to the type constructor are: type(name, bases, dct) - name is a string giving the name of the class to be constructed - bases is a tuple giving the parent classes of the class to be constructed - dct is a dictionary of the attributes and methods of the class to be constructed

So, for example, the following two pieces of code have identical results:

```
class Foo(object): i = 4
```

```
class Bar(Foo): def get_i(self) : return self.i
```

```
b = Bar() print(b.get_i()) 4 Foo = type('Foo', (), dict(i = 4))
```

```
Bar = type('Bar', (Foo,), dict(get_i = lambda self : self.i))
```

b = Bar() print(b.get_i()) 4 *This perhaps seems a bit over-complicated in the case of this contrived example, but it is the - fly.*

Making Things Interesting: Custom Metaclasses Now things get really fun. Just as we can inherit from and extend a class we've created, we can also inherit from and extend the type metaclass, and create custom behavior in our metaclass.

Example 1: Modifying Attributes Let's use a simple example where we want to create an API in which the user can create a set of interfaces which contain a file object. Each interface should have a unique string ID, and contain an open file object. The user could then write specialized methods to accomplish certain

tasks. There are certainly good ways to do this without delving into metaclasses, but such a simple example will (hopefully) elucidate what's going on. First we'll create our interface meta class, deriving from type:

```
class InterfaceMeta(type):
    def __new__(cls, name, parents, dct):
        createaclass_if_it's not specified if class_id not in dct:
            dct['class_id'] = name.lower()
        open the specified file for writing if 'file' in dct:
            filename = dct['file']
            dct['file'] = open(filename, 'w')
        we need to call type.__new__ to complete the initialization
        return super(InterfaceMeta, cls).__new__(cls, name, parents, dct)
        Notice that we've modified
```

Now we'll use our InterfaceMeta class to construct and instantiate an Interface object:

```
Interface = InterfaceMeta('Interface', (), dict(file='tmp.txt'))
print(Interface.class_id)
print(Interface.file)
interface < open file 'tmp.txt', mode 'w' at 0x21b8810 >
This behaves as we'd expect: the class_id class variable is created, and the file class variable is replaced with an open
class Interface(object):
    __metaclass__ = InterfaceMeta
    file = 'tmp.txt'
print(Interface.class_id)
print(Interface.file)
interface < open file 'tmp.txt', mode 'w' at 0x21b8ae0 >
by defining the __metaclass__ attribute of the class, we've told the class that it should be constructed using InterfaceMeta rather than using type.
To make type(Interface) return InterfaceMeta
Furthermore, any class derived from Interface will now be constructed using the same metaclass:
class UserInterface(Interface):
    file = 'foo.txt'
print(UserInterface.file)
print(UserInterface.class_id)
< open file 'foo.txt', mode 'w' at 0x21b8c00 >
user interface
This simple example shows how metaclasses can be used to create powerful and flexible APIs for
```

Example 2: Registering Subclasses Another possible use of a metaclass is to automatically register all subclasses derived from a given base class. For example, you may have a basic interface to a database and wish for the user to be able to define their own interfaces, which are automatically stored in a master registry.

You might proceed this way:

```
class DBInterfaceMeta(type):
    def __new__(cls, name, bases, dct):
        we use __init__ rather than __new__ here because we want to modify attributes of the class *after* they have been created
        super(DBInterfaceMeta, cls).__init__(name, bases, dct)
        Our metaclass simply adds a registry dictionary if it's not already present, and adds the new
class DBInterface(object):
    __metaclass__ = DBInterfaceMeta
    registry = {}
    Now let's create some subclasses, and double-check
    that they're added to the registry:
class FirstInterface(DBInterface):
    pass
class SecondInterface(DBInterface):
    pass
class SecondInterfaceModified(SecondInterface):
    pass
print(DBInterface.registry)
{'firstinterface': <class 'main.FirstInterface'>, 'secondinterface': <class 'main.SecondInterfaceModified'>, 'secondinterface': <class 'main.SecondInterface'>, 'secondinterface': <class 'main.SecondInterface'>}
```

Conclusion: When Should You Use Metaclasses? I've gone through some examples of what metaclasses are, and some ideas about how they might be used to create very powerful and flexible APIs. Although metaclasses are in the background of everything you do in Python, the average coder rarely has to think about them.

But the question remains: when should you think about using custom metaclasses in your project? It's a complicated question, but there's a quotation floating around the web that addresses it quite succinctly:

Metaclasses are deeper magic than 99% of users should ever worry about. If you wonder whether you need them, you don't (the people who actually need them know with certainty that they need them, and don't need an explanation about why).

Tim Peters

In a way, this is a very unsatisfying answer: it's a bit reminiscent of the wistful and clichéd explanation of the border between attraction and love: "well, you just... know!"

But I think Tim is right: in general, I've found that most tasks in Python that can be accomplished through use of custom metaclasses can also be accomplished more cleanly and with more clarity by other means. As programmers, we should always be careful to avoid being clever for the sake of cleverness alone, though it is admittedly an ever-present temptation.

I personally spent six years doing science with Python, writing code nearly on a daily basis, before I found a problem for which metaclasses were the natural solution. And it turns out Tim was right:

I just knew.

2.2 Hệ điều hành

2.2.1 File Operations

Copy folder

```
import shutil
shutil.copyfile("src", "dst")
```

2.2.2 CLI

shutil - High-level file operations

2.3 Cơ sở dữ liệu (chưa xây dựng)

2.4 Giao diện (chưa xây dựng)

2.5 Lập trình mạng

REST JSON 1 2 GET

```
import requests
url = "http://localhost:8080/messages"
response = requests.get(url)
data = response.json()
```

POST

```
import requests
import json

url = "http://localhost:8080/messages"
data = {'sender': 'Alice', 'receiver': 'Bob', 'message': 'Hello!'}
headers = {
    'Content-type': 'application/json',
    'Accept': 'application/json'
}
r = requests.post(url, data=json.dumps(data), headers=headers)
```

Chương 3

Phát triển phần mềm với Python

3.0.1 Mục tiêu của khóa học

3.0.2 Đối tượng học viên

- Đã lập trình Python được 1-2 năm
- Muốn phát triển phần mềm mã nguồn mở

3.1 Logging

levels, attributes references

The logging library takes a modular approach and offers several categories of components: loggers, handlers, filters, and formatters.

Loggers expose the interface that application code directly uses. Handlers send the log records (created by loggers) to the appropriate destination. Filters provide a finer grained facility for determining which log records to output. Formatters specify the layout of log records in the final output. Step 0: Project structure

```
code/  
  main.py  
  config  
    logging.conf  
  logs  
    app.log
```

Step 1: Create file logging.conf

```
[loggers] keys=root
```

```
[handlers] keys=consoleHandler,fileHandler
```

```
[formatters] keys=formatter
```

```
[logger,oot]level = DEBUGhandlers = consoleHandler, fileHandler
```

```
[handler_consoleHandler]class = StreamHandlerlevel = DEBUGformatter =  
formatterargs = (sys.stdout,)
```

```
[handler_fileHandler]class = FileHandlerlevel = DEBUGformatter = formatterargs =
('logs/app.log','a')
[formatter_formatter]format = datefmt = Step2 : Loadconfigandcreatelogger
In main.py
import logging.config
load logging config logging.config.fileConfig('config/logging.conf') Step 3: In
your application code
logging.getLogger().debug('debug message') logging.getLogger().info('info mes-
sage') logging.getLogger().warn('warn message') logging.getLogger().error('error
message') logging.getLogger().critical('critical message') More Resources
Introduction to Logging Quick and simple usage of python log Python: Logging
module
Python: Logging cookbook
Python: Logging guide
```

3.2 Configuration

```
pyconfiguration
Installation conda install -c rain1024 pyconfiguration Usage Step 1: Create con-
fig.json file
"SERVICE_URL" : "http://api.service.com" Step2 : Addthesecodetomain.pyfile
from pyconfiguration import Configuration Configuration.load('config.json') print
Configuration.SERVICE_URL
> http://api.service.com References: What's the best practice using a settings
file 1
What's the best practice using a settings file in Python?
```

3.3 Command Line

Command Line Arguments There are the following modules in the standard library:

The getopt module is similar to GNU getopt. The optparse module offers object-oriented command line option parsing. Here is an example that uses the latter from the docs:

```
from optparse import OptionParser
parser = OptionParser() parser.add_option("-f", "--file", dest = "filename", help =
"write report to FILE", metavar = "FILE") parser.add_option("-q", "--quiet", action =
"store_false", dest = "verbose", default = True, help = "don't print status messages to stdout")
(options, args) = parser.parse_args() optparse supports (among other things) :
Multiple options in any order. Short and long options. Default values. Gener-
ation of a usage help message. Suggest Reading Command Line Arguments In
Python
```

3.4 Testing

Testing your code is very important.

Getting used to writing testing code and running this code in parallel is now considered a good habit. Used wisely, this method helps you define more precisely your code's intent and have a more decoupled architecture.

Unittest unittest is the batteries-included test module in the Python standard library. Its API will be familiar to anyone who has used any of the JUnit/nUnit/CppUnit series of tools.

The Basics Creating test cases is accomplished by subclassing unittest.TestCase.

```
import unittest
```

```
def fun(x): return x + 1
```

```
class MyTest(unittest.TestCase):
    def test(self):
        self.assertEqual(fun(3), 4)
    # Skipping tests
    # Unittest supports skipping individual test methods and even whole
    # classes of tests. In addition, it supports marking a test as an "expected failure,"
    # a test that is broken and will fail, but shouldn't be counted as a failure on a
    # code TestResult.
```

Skipping a test is simply a matter of using the skip() decorator or one of its conditional variants.

```
import sys
import unittest
```

```
class MyTestCase(unittest.TestCase):
```

```
@unittest.skip("demonstrating skipping")
def test_nothing(self):
    self.fail("shouldn't happen")
```

```
@unittest.skipIf(mylib.__version__ < (1,3),
    "not supported in this library version")
def test_format(self):
    # Tests that work for only a certain version of
```

```
@unittest.skipUnless(sys.platform.startswith("win"),
    "requires Windows")
def test_windows_support(self):
    # windowsspecific testing code
    pass
Tox is a generic virtualenv management and test command line tool you can
```

use for:

checking your package installs correctly with different Python versions and interpreters running your tests in each of the environments, configuring your test tool of choice acting as a frontend to Continuous Integration servers, greatly reducing boilerplate and merging CI and shell-based testing. Installation

You can install tox with pip using the following command

```
> pip install tox
```

Setup default environment in Windows with conda

```
> conda create -p C:\python27 python=2.7
```

```
> conda create -p C:\python34 python=3.4
```

Related Readings Testing Your Code, The Hitchhiker's Guide to Python unittest Unit testing framework, docs.python.org Is it possible to use tox with conda-based Python installations?, stackoverflow

3.5 IDE & Debugging

Today, I write some notes about my favorite Python IDE - PyCharm. I believe it's a good one for developing python, which supports git, vim, etc. This list below contains my favorite features.

Pycharm Features Intelligent Editor Navigation Graphical Debugger Refactorings Code Inspections Version Control Integration Scientific Tools Intelligent Editor PyCharm provides smart code completion, code inspections, on-the-fly error highlighting and quick-fixes, along with automated code refactorings and rich navigation capabilities.

Syntax Highlighting

Read your code easier with customizable colors for Python code and Django templates. Choose from several predefined color themes.

Auto-Indentation and code formatting

Automatic indents are inserted on new line. Indent verification and code re-formatting are compliant with project code-style settings.

Configurable code styles

Select a predefined coding style to apply to your code style configuration for various supported languages.

Code completion

Code completion for keywords, classes, variables, etc. as you type or via Ctrl+Space.

Editor suggestions are context-aware and offer the most appropriate options.

Keyboard shortcuts: Tab, Alt+Enter

Code selection and comments

Select a block of code and expand it to an expression, to a line, to a logical block of code, and so on with shortcuts. Single keystroke to comment/uncomment the current line or selection.

Code formatter

Code formatter with code style configuration and other features help you write neat code that's easy to support. PyCharm contains built-in PEP-8 for Python and other standards compliant code formatting for supported languages.

Code snippets and templates

Save time using advanced customizable and parametrized live code templates and snippets.

Keyboard shortcuts check if ENTER

if check: type something *Code folding*

Code folding, auto-insertion of braces, brackets, quotes, matching brace/bracket highlighting, etc.

On-the-fly error highlighting

Errors are shown as you type. The integrated spell-checker verifies your identifiers and comments for misspellings.

Multiple carets and selections

With multiple carets, you can edit several locations in your file at the same time.

Keyboard shortcuts: SHIFT + F6

Code analysis

Numerous code inspections verify Python code as you type and also allow inspecting the whole project for possible errors or code smells.

Quick-fixes

Quick-fixes for most inspections make it easy to fix or improve the code instantly.

Alt+Enter shows appropriate options for each inspection.

Keyboard shortcuts: F2

Duplicated code detector

Smart duplicated code detector analyzes your code and searches for copy/pasted code. You'll be presented with a list of candidates for refactoring and with the help of refactorings it's easy to keep your code dry.

Configurable language injections

Natively edit non-Python code embedded into string literals, with code completion, error-highlighting, and other coding assistance features.

Code auto generation

Code auto-generation from usage with quick-fixes; docstrings and the code matching verification, plus autoupdate on refactoring. Automatic generation of a docstring stub (reStructuredText, Epytext, Google, and NumPy).

Intention actions

Intention actions help you apply automated changes to code that is correct, to improve it or to make your coding routine easier.

Searching

Keyboard shortcuts: Double Shift (search everywhere)

Navigation Shortcuts

Keyboard shortcuts: ALT + SHIFT + UP/DOWN (move line up and down)

Graphical Debugger PyCharm provides extensive options for debugging your Python/Django and JavaScript code:

Set breakpoints right inside the editor and define hit conditions Inspect context-relevant local variables and user-defined watches, including arrays and complex objects, and edit values on the fly Set up remote debugging using remote interpreters Evaluate an expression in runtime and collect run-time type statistics for better autocompletion and code inspections Attach to a running process Debug Django templates

Inline Debugger

With an inline debugger, all live debugging data are shown directly in the editor, with variable values integrated into the editor's look-and-feel. Variable values can be viewed in the source code, right next to their usages.

Step into My Code

Use Step into My Code to stay focused on your code: the debugger will only step through your code bypassing any library sources.

Multi-process debugging

PyCharm can debug applications that spawn multiple Python processes, such as Django applications that don't run in -no-reload mode, or applications using many other Web frameworks that use a similar approach to code auto-reloading.

Run/Debug configurations

Every script/test or debugger execution creates a special 'Run/Debug Configuration' that can be edited and used later. Run/Debug Configurations can be shared with project settings for use by the whole team.

Workspace Custom Scheme Go to File - Settings... then Editor - Colors Fonts

Now you can change your scheme, I like Darcular

https://confluence.jetbrains.com/download/attachments/51945983/appearance3.png?version=1modificationDate=1519459830000&_af=1

IPython Support PyCharm supports usage of IPython magic commands.

<http://i.stack.imgur.com/aTEW2.png>

Vim Support You can configure PyCharm to work as a Vim editor

https://confluence.jetbrains.com/download/attachments/51946537/vim4.png?version=1modificationDate=1519465370000&_af=1

Keyboard Shortcuts: Ctrl+Shift+V (paste)

3.5.1 Pycharm Pycharm

Hôm nay tự nhiên lại gặp lỗi không tự nhận unittest, không resolve được package import bởi relative path. Vụ không tự nhận unittest sửa bằng cách xóa file .idea là xong. Còn vụ không resolve được package import bởi relative path thì vẫn chịu rồi. Nhìn code cứ đở lờm khó chịu thật.

01/2018: Pycharm là trình duyệt ưa thích của mình trong suốt 3 năm vừa rồi.

3.6 Package Manager

3.6.1 py2exe

py2exe is a Python Distutils extension which converts Python scripts into executable Windows programs, able to run without requiring a Python installation.

Installation

```
# py2exe
conda install -c https://conda.anaconda.org/clinicalgraphics cg-py2exe
Build 1
python setup.py py2exe
# build PyQT
python setup.py py2exe --includes sip
```

Known Issues

Error: Microsoft Visual C++ 10.0 is required (Unable to find vcvarsall.bat) (link)

How to fix

Step 1: Install Visual Studio 2015

Step 2:

```
set VS100COMNTOOLS=%VS140COMNTOOLS%
```

3.6.2 Quản lý gói với Anaconda

Cài đặt package tại một branch của một project trên github

```
> pip install git+https://github.com/tangentlabs/django-oscar-paypal.
↪ git@issue/34/oscar-0.6#egg=django-oscar-paypal
```

Trích xuất danh sách package

```
> pip freeze > requirements.txt
```

Chạy ipython trong environment anaconda

Chạy dòng lệnh này

```
conda install nb_conda
source activate my_env
python -m IPython kernelspec install-self --user
ipython notebook
```

Interactive programming với ipython

Trích xuất ipython ra slide (không hiểu sao default ‘–to slides’ không work nữa, lại phải thêm tham số ‘–reveal-prefix’¹

```
jupyter nbconvert "file.ipynb"
--to slides
--reveal-prefix "https://cdnjs.cloudflare.com/ajax/libs/reveal.js/3.1.0"
```

¹<https://github.com/jupyter/nbconvert/issues/91#issuecomment-283736634>

****Tham khảo thêm****

* <https://stackoverflow.com/questions/37085665/in-which-conda-environment-is-jupyter-executing> * <https://github.com/jupyter/notebook/issues/541#issuecomment-146387578> * <https://stackoverflow.com/a/20101940/772391>

3.7 Environment

Similar to pip, conda is an open source package and environment management system ¹. Anaconda is a data science platform that comes with a lot of packages. It uses conda at the core. Unlike Anaconda, Miniconda doesn't come with any installed packages by default. Note that for miniconda, everytime you open up a terminal, conda won't automatically be available. Run the command below to use conda within miniconda.

Conda Let's first start by checking if conda is installed.

```
> conda --version
```

```
conda 4.2.12
```

To see the full documentation **for** any **command**, type the **command**

→ followed by **--help**. For example, to learn about the conda update

→ **command**:

```
> conda update --help
```

Once it has been confirmed that conda has been installed, we will now

→ make sure that it is up to date.

```
> conda update conda
```

Using Anaconda Cloud api site <https://api.anaconda.org>

Fetching package metadata:

.Solving package specifications:

Package plan **for** installation **in** environment `//anaconda`:

The following packages will be downloaded:

package	build	
conda- env -2.6.0	0	601 B
ruamel_yaml-0.11.14	py27_0	184 KB
conda-4.2.12	py27_0	376 KB
Total:		560 KB

The following NEW packages will be INSTALLED:

ruamel_yaml: 0.11.14-py27_0

The following packages will be UPDATED:

```

conda:      4.0.7-py27_0 --> 4.2.12-py27_0
conda-env:  2.4.5-py27_0 --> 2.6.0-0
python:     2.7.11-0    --> 2.7.12-1
sqlite:     3.9.2-0     --> 3.13.0-0

Proceed ([y]/n)? y

Fetching packages ...
conda-env-2.6. 100% |#
  ↳ #####| Time:
  ↳ 0:00:00 360.78 kB/s
ruamel_yaml-0. 100% |#
  ↳ #####| Time:
  ↳ 0:00:00 5.53 MB/s
conda-4.2.12-p 100% |#
  ↳ #####| Time:
  ↳ 0:00:00 5.84 MB/s
Extracting packages ...
[ COMPLETE ]|#
  ↳ #####|
  ↳ 100%
Unlinking packages ...
[ COMPLETE ]|#
  ↳ #####|
  ↳ 100%
Linking packages ...
[ COMPLETE ]|#
  ↳ #####|
  ↳ 100%
Environments

```

3.7.1 Create

In order to manage environments, we need to create at least two so you can move or switch between them. To create a new environment, use the conda create command, followed by any name you wish to call it:

```

# create new environment
conda create -n <your_environment> python=2.7.11

```

3.7.2 Clone

Make an exact copy of an environment by creating a clone of it. Here we will clone snowflakes to create an exact copy named flowers:

```

conda create --name flowers --clone snowflakes

```

3.7.3 List

List all environments

Now you can use conda to see which environments you have installed so far. Use the conda environment info command to find out

```
> conda info -e

conda environments:
snowflakes      /home/username/miniconda/envs/snowflakes
bunnies         /home/username/miniconda/envs/bunnies
```

Verify current environment

Which environment are you using right now - snowflakes or bunnies? To find out, type the command:

```
conda info --envs
```

3.7.4 Remove

If you didn't really want an environment named flowers, just remove it as follows:

```
conda remove --name flowers --all
```

3.7.5 Share

You may want to share your environment with another person, for example, so they can re-create a test that you have done. To allow them to quickly reproduce your environment, with all of its packages and versions, you can give them a copy of your environment.yml file.

Export the environment file

To enable another person to create an exact copy of your environment, you will export the active environment file.

```
conda env export > environment.yml
```

Use environment from file

Create a copy of another developer's environment from their environment.yml file:

```
conda env create -f environment.yml
# remove environment
conda remove -n <your _environemnt> --all
```

3.8 Module

Create Public Module conda, pypi, github

Step 0/4: Check your package name Go to https://pypi.python.org/pypi/your_package_name_to_see_your_package_name

Step 1/4: Make your module 1 1.1 pip install cookiecutter

1.2 cookiecutter <https://github.com/audreyr/cookiecutter-pypackage.git>

1.3 Fill all necessary information

full_name[AudreyRoyGreenfeld] : email[aroy@alum.mit.edu] : github_username[audreyr] :

project_name[PythonBoilerplate] : project_slug[] : project_short_description :

release_date[] : pypi_username[] : year[2016] : version[0.1.0] : use_pypi_deployment_with_travis[y] :

Itwillcreateadirectory

```

|- LICENSE |- README.md |- TODO.md |- docs | |- conf.py | |- generated | |-
index.rst | |- installation.rst | |- modules.rst | |- quickstart.rst | |- sandman.rst |
requirements.txt |- your_package | |- init.py | |- your_package.py | |- test | |- models.py | |- test_your_package.py | |- setup.py Step 2/4: G
2. Create a .pypirc configuration file in HOME directory
[distutils] index-servers = pypi
[pypi] repository=https://pypi.python.org/pypi username=your_username password =
your_password 3. Change your MANIFEST.in
recursive-include project_folder * 4. Upload your package to PyPI
python setup.py register -r pypi python setup.py sdist upload -r pypi Step 4/4:
Conda 2 1. Install conda tools
conda install conda-build conda install anaconda-client 2. Build a simple package
with conda skeleton pypi
cd your_package_folder mkdir conda_skeleton pypi your_package This creates a directory named your_package
|- your_package | |- bld.bat | |- meta.yaml | |- build.sh 3. Build your package
conda build your_package
convert to all platform conda convert -f -platform all C:-bld-64_package -0.1.1 -
py270.tar.bz2 4. Upload package to Anaconda
anaconda login anaconda upload linux-32/your_package.tar.bz2 anaconda upload linux-
64/your_package.tar.bz2 anaconda upload win-32/your_package.tar.bz2 anaconda upload win-
64/your_package.tar.bz2 Create Private Module Step 1: Make your module 1.1 pip install cookiecutter
1.2 cookiecutter https://github.com/audreyr/cookiecutter-pypackage.git
1.3 Fill all necessary information
full_name[AudreyRoyGreenfeld] : email[aroy@alum.mit.edu] : github_username[audreyr] :
project_name[PythonBoilerplate] : project_slug[] : project_short_description :
release_date[] : pypi_username[] : year[2016] : version[0.1.0] : use_pypi_deployment_with_travis[y] :
Step 2: Build your module Change your MANIFEST.in
recursive-include project_folder * Build your module with setup.py
cd your_project_folder
build local python setup.py build > It will create a new folder in > PYTHON_HOME/Lib/sites-
packages/your_project_name - 0.1.0 - py2.7.egg
build distribution python setup.py sdist > It will create a zip file in PROJECT_FOLDER/dist Step 3 :
Usage your module In the same machine
import your_project_name In other machine
Python: Build Local Package with Conda Here is a step by step tutorial
about building a local module package install it from a custom channel 1
Step 1: Make a setup folder for your package with cookiecutter on terminal:
mkdir build cd build pip install cookiecutter cookiecutter https://github.com/audreyr/cookiecutter-
pypackage.git
Fill all necessary information
full_name[AudreyRoyGreenfeld] : email[aroy@alum.mit.edu] : github_username[audreyr] :
project_name[PythonBoilerplate] : project_slug[] : project_short_description :
release_date[] : pypi_username[] : year[2016] : version[0.1.0] : use_pypi_deployment_with_travis[y] :
It will create a directory
|- LICENSE |- README.md |- TODO.md |- docs | |- conf.py | |- generated | |-
index.rst | |- installation.rst | |- modules.rst | |- quickstart.rst | |- sandman.rst |
requirements.txt |- your_package | |- init.py | |- your_package.py | |- test | |- models.py | |- test_your_package.py | |- setup.py Copy your
Add this line to MANIFEST.in
recursive-include project_folder * Step 2: Build conda package mkdir conda_channel git clone https://
//github.com/hunguyen1702/condaBuildLocalTemplate.git mv condaBuildLocalTemplate/your_package_name

```



```

rf.gitREADME.md Edit the file meta.yaml with the instruction inside it
cd .. conda build your_package_name
Step 1: Create custom channel and install from local package
Create a channel directory
cd channel
Convert your_package you've built to all platform
conda convert -platform all /anaconda/conda-bld/linux-64/your_package_0.1.0-
py27_0.tar.bz2 and this will create :
channel/ linux-64/ package-1.0-0.tar.bz2 linux-32/ package-1.0-0.tar.bz2 osx-
64/ package-1.0-0.tar.bz2 win-64/ package-1.0-0.tar.bz2 win-32/ package-1.0-
0.tar.bz2
Register your package to your new channel
cd .. conda index channel/linux-64 channel/osx-64 channel/win-64
Verify your new channel
conda search -c file://path/to/channel/ --override-channels
If you see your_package's appearance, so it's worked
After that if you want to install that package from local, run this command:
conda install --use-local your_package
and when you want to create environment with local package from file, you just
have export environment to .yaml file and add this channels section before the
dependencies section:
channels: - file://path/to/your/channel/

```

3.9 Production

```

Production with docker Base Image: magizbox/conda2.7/
Docker Folder
your_app/appconfig/main.py Docker file run.sh Docker file
FROM magizbox/conda2.7:4.0
ADD ./app /app ADD ./run.sh /run.sh
RUN conda env create -f environment.yml run.sh
source activate your_environment
cd /app
python main.py
Compose
service: build: ./service-app command: 'bash run.sh'
Note: an other python conda with lower version (such as 3.5), will occur error when install requests
package
python 3.4 hay 3.5
Có lẽ 3.5 là lựa chọn tốt hơn (phải có của tensorflow, pytorch, hỗ trợ mock)
Quản lý môi trường phát triển với conda
Chạy lệnh 'remove' để xóa một môi trường

```

```
conda remove --name flowers --all
```

3.10 Test với python

Sử dụng những loại test nào?

Hiện tại mình đang viết unittest với default class của python là unittest. Thực ra toàn sử dụng 'assertEqual' là chính!

Ngoài ra mình cũng đang sử dụng tox để chạy test trên nhiều phiên bản python (python 2.7, 3.5). Điều hay của tox là mình có thể thiết kế toàn bộ cài đặt project và các dependencies package trong file 'tox.ini'

Chạy test trên nhiều phiên bản python với tox

Pycharm hỗ trợ debug tox (quá tuyệt!), chỉ với thao tác đơn giản là nhấn chuột phải vào file tox.ini của project.

3.11 Xây dựng docs với readthedocs và sphinx

20/12/2017: Tự nhiên hôm nay tất cả các class có khai báo kế thừa ở project languageflow không thể index được. Vài thật. Làm thẳng đê không biết đâu mà build model.

Thử build lại chục lần, thay đổi file conf.py và package_reference.rst chán chê không được. Giả thiết đầu tiên là do hai nguyên nhân (1) docstring ghi sai, (2) nội dung trong package_reference.rst bị sai. Sửa chán chê cũng vẫn thế, thử checkout các commit của git. Không hoạt động!

Mất khoảng vài tiếng mới để ý thẳng readthedocs có phần log cho từng build một. Lần mò vào build gần nhất và build (mình nhớ là) thành công cách đây 2 ngày

Log build gần nhất

```
Running Sphinx v1.6.5
making output directory...
loading translations [en]... done
loading intersphinx inventory from https://docs.python.org/objects.inv...
intersphinx inventory has moved: https://docs.python.org/objects.inv ->
  ↪ https://docs.python.org/2/objects.inv
loading intersphinx inventory from http://docs.scipy.org/doc/numpy/
  ↪ objects.inv...
intersphinx inventory has moved: http://docs.scipy.org/doc/numpy/
  ↪ objects.inv -> https://docs.scipy.org/doc/numpy/objects.inv
building [mo]: targets for 0 po files that are out of date
building [readthedocsdirhtml]: targets for 8 source files that are out of
  ↪ date
updating environment: 8 added, 0 changed, 0 removed
reading sources... [ 12%] authors
reading sources... [ 25%] contributing
reading sources... [ 37%] history
reading sources... [ 50%] index
reading sources... [ 62%] installation
reading sources... [ 75%] package_reference
reading sources... [ 87%] readme
reading sources... [100%] usage

looking for now-outdated files... none found
pickling environment... done
checking consistency... done
preparing documents... done
writing output... [ 12%] authors
writing output... [ 25%] contributing
writing output... [ 37%] history
writing output... [ 50%] index
writing output... [ 62%] installation
writing output... [ 75%] package_reference
```

```
writing output... [ 87%] readme
writing output... [100%] usage
```

Log build hồi trước

```
Running Sphinx v1.5.6
making output directory...
loading translations [en]... done
loading intersphinx inventory from https://docs.python.org/objects.inv...
intersphinx inventory has moved: https://docs.python.org/objects.inv ->
  ↪ https://docs.python.org/2/objects.inv
loading intersphinx inventory from http://docs.scipy.org/doc/numpy/
  ↪ objects.inv...
intersphinx inventory has moved: http://docs.scipy.org/doc/numpy/
  ↪ objects.inv -> https://docs.scipy.org/doc/numpy/objects.inv
building [mo]: targets for 0 po files that are out of date
building [readthedocs]: targets for 8 source files that are out of date
updating environment: 8 added, 0 changed, 0 removed
reading sources... [ 12%] authors
reading sources... [ 25%] contributing
reading sources... [ 37%] history
reading sources... [ 50%] index
reading sources... [ 62%] installation
reading sources... [ 75%] package_reference
reading sources... [ 87%] readme
reading sources... [100%] usage

/home/docs/checkouts/readthedocs.org/user_builds/languageflow/
  ↪ checkouts/develop/languageflow/transformer/count.py:docstring
  ↪ of languageflow.transformer.count.CountVectorizer:106:
  ↪ WARNING: Definition list ends without a blank line; unexpected
  ↪ unindent.
/home/docs/checkouts/readthedocs.org/user_builds/languageflow/
  ↪ checkouts/develop/languageflow/transformer/tfidf.py:docstring of
  ↪ languageflow.transformer.tfidf.TfidfVectorizer:113: WARNING:
  ↪ Definition list ends without a blank line; unexpected unindent.
../README.rst:7: WARNING: nonlocal image URI found: https://img.
  ↪ shields.io/badge/latest-1.1.6-brightgreen.svg
looking for now-outdated files... none found
pickling environment... done
checking consistency... done
preparing documents... done
writing output... [ 12%] authors
writing output... [ 25%] contributing
writing output... [ 37%] history
writing output... [ 50%] index
writing output... [ 62%] installation
writing output... [ 75%] package_reference
writing output... [ 87%] readme
writing output... [100%] usage
```

Đập vào mắt là sự khác biệt giữa documentation type

Lỗi

```
building [readthedocsdirhtml]: targets for 8 source files that are out of
    ↪ date
```

Chạy

```
building [readthedocs]: targets for 8 source files that are out of date
```

Hí ha hí hửng. Chắc trong cơn bất loạn sửa lại settings đây mà. Sửa lại nó trong phần Settings (Admin gt; Settings gt; Documentation type)

Khi chạy nó đã cho ra log đúng

```
building [readthedocsdirhtml]: targets for 8 source files that are out of
    ↪ date
```

Nhưng vẫn lỗi. Vãi!!! Sau khoảng 20 phút tiếp tục bắn loạn, chửi bới readthedocs các kiểu. Thì để ý dòng này

Lỗi

```
Running Sphinx v1.6.5
```

Chạy

```
Running Sphinx v1.5.6
```

Ngay dòng đầu tiên mà không để ý, ngu thật. Aha, Hóa ra là thằng readthedocs nó tự động update phiên bản sphinx lên 1.6.5. Mình là mình chúa ghét thay đổi phiên bản (code đã mệt rồi, lại còn phải tương thích với nhiều phiên bản nữa thì ăn c** à). Đầu tiên search với Pycharm thấy dòng này trong ‘conf.py’

```
# If your documentation needs a minimal Sphinx version, state it here.
# needs_sphinx = '1.0'
```

Đổi thành

```
# If your documentation needs a minimal Sphinx version, state it here.
needs_sphinx = '1.5.6'
```

Vẫn vậy (holy sh*t). Thử sâu một tạo (thực sự là rất nhiều tạo). Thấy cái này trong trang Settings

Ờ há. Thằng đàn này cho phép trỏ đường dẫn tới một file trong project để cấu hình dependency. Haha. Tạo thêm một file ‘requirements’ trong thư mục ‘docs’ với nội dung

```
sphinx==1.5.6
```

Sau đó cấu hình nó trên giao diện web của readthedocs

Build thử. Build thử thôi. Cảm giác đúng lắm rồi đấy. Và... nó chạy. Ahihi

Kinh nghiệm

* Khi không biết làm gì, hãy làm 3 việc. Đọc LOG. Phân tích LOG. Và cố gắng để LOG thay đổi theo ý mình.

PS: Trong quá trình này, cũng không thêm build thành PDF với Epub nữa. Tiết kiệm được bao nhiêu thời gian.

Tài liệu tham khảo