LEARNING PYTHON, PART 4: ADVANCED PROGRAMMING

1. Data Streams

Iterables and iterators

- Iterables: data types such as list or tuple
- Changes iterbales to iterators

```
lst_iter = lst.__iter__()
```

- Step through each element in an iterator

```
## Python 2
next(lst_iter) ## also works in Python 3
lst_iter.next()

## Python 3
__next__(lst_iter)  ## or
lst_iter.__next__()
```

Generators

- Generators can be used to maximize memory efficiency:
  - Items are loaded into memory only as needed.
  - Items are not saved when the generator is advanced to the next item.

- Can be used in the `for` loop

```
## Non-generator version
for i in [1, 2, 3, 4, 5]:
    pass;
## Generator version (recommended!)
for i in (1, 2, 3, 4 5):
    pass;
```

- Can be used in the list comprehension, wrapped around by parentheses.

```
## suppose x is a list or a tuple, then
x_gen = (formula_i for i in x if condition)
```

- In Python 2:
  `xrange(n)`: the generator counterpart of `range(n)` ; can be used in the `for` loop or list comprehension.
  In Python 3:
  `range(n)` is a generator.

Reading (large) files

- Use `open()` function
- Use `.rstrip()` to remove the newlines
- Use `.split(',')` to separate the columns

```
with open(filename, 'r') as handler:
    for line in handler:
        print([int(i) for i in line.rstrip().split(',')])
```

Creating generators

We can define a function, for example

```
## Define a function
def count_by_n(n):
    for i in xrange(5):
        yield n * i
## Create a generator of (0, 7, 14, 21, 28)
x_gen = count_by_n(7)
```

Note: Every time the `next()` function is called on a generator, the generator runs until it reaches a `yield` statement, returns that value, and then waits.

2. Data Structure: Classes

Defining a class

For example, called `Xclass`

```
class Xclass(object):
    def __init__(self, attribute1, attribute2):
        self.attribute1 = attribute1
        self.attribute2 = attribute2

    def func(self):
       return self.attribute1 * self.attribute2
```

Note: In Python 2, remember to add the `(object)` when defining the class; while in Python 3, it can be omitted since it's assumed.

- Create a class

```
## Create a class
xc1 = Xclass(value1, value2)

## Access class attributes
xc1.attribute1 ## returns value1

## Apply class functions
xc1.func()
```

Change an attribute in a class

```
## Change an attribute in a class
xc1.attribute1 = new_value
```

However, this should be used with caution, because other attributes may depend on `attribute1`, but changing `attribute1` won't automatically change others.

So in order to avoid the above change of value, we can use double underscore `__attribute1` when defining the class, and also write functions to view and change this attribute.

```
class Xclass(object):
    def __init__(self, attribute1, attribute2):
        self.__attribute1 = attribute1
        self.attribute2 = attribute2

    def get_attribute1(self):
       return self.__attribute1

    def set_attribute1(self, new_value)
       self.__attribute1 = new_value
```

Note that in this case

```
xc1.__attribute1 ## returns an error of no such attibutes
xc1.get_attribute1() ## returns the value of __attribute1
xc1.set_attribute1(new_value) ## changes the value of __attribute1
```

Super and sub classes

In the following example, `Dog` is a sub-class of `Animal`, and `Animal` is a sub-class of `object`.

```
class Animal(object):
    def __init__(self,age,price):
        self.age = age
        self.price = price

class Dog(Animal):
    def __init__(self,age, price, breed):
        self.breed = breed
        super(Dog,self).__init__(age, price)
```

Notes:
(1) The function `super` retrieves the parent of the current class `Dog`, which is `Animal`. So this let us to call methods of the parent class.

- Check if an object is a member of a class

```
isinstance(object, class) ## returns True or False
```

The use of `*args` and `**kwargs`

When defining functions

To pass *an arbitrary number* of variables.

- `*args` passes non-keyworded arguments into a tuple.
- `**kwargs` passes keyworded arguments into a dictionary

```
def foo(*args, **kwargs): ##
    print('args = {}'.format(args))
    print('kwargs = {}'.format(kwargs))
    for item in kwargs: ## item is each keyword in kwargs
        print('item = {0}: {1}'.format(item, kwargs[item]))

foo(4, 5, 6, a=1, b=2)
## returns:
## args = (4, 5, 6)
## kwargs = {'a': 1, 'b': 2}
## item = a: 1
## item = b: 2
```

Notes:
(1) It's `*` and `**` that are important; whether using `args` `kwargs` or other names doesn't matter.
(2) `*args` must be before `**kwargs`

When call functions

To unpack a list or dictionary into arguments

- `*args` unpack a list into multiple arguments.
- `**kwargs` unpack a dictionary into keyworded arguments.

3. Writing Good Codes

Assertions

An assertions is to make sure something must be true at a certain point in the program. For example, when defining a function, to make sure a certain argument is always positive, otherwise, the traceback (error message) will say "AssertionError", followed by the reason stated in the `assert` function.

```
def foo(balance):
    assert balance > 0, 'balance must be positive.'
    pass
```

Two rules of adding assertions:

1. Good code catches mistakes as early as possible.
2. Turn bugs in to assertions or tests.

Exceptions

If any of the code in the `try` block yields an error (of the `ErrorType`), then run the `except` block.

```
try:
    block of code
except ErrorType:
    block of code
```

Note: `ErrorType` is optional (but recommended); the common ones are `IOError`, `SyntaxError`, `NameError`, `IndexError`, `KeyError`, `Exception`.

4. Modules and Packages (i.e., Libraries)

A module is a python file, and a package is a folder containing python files. Either can be referred to as a library.

General rules for file and module names:

- A module name is all lower case.
- A file name starts with a lower case letter, and continues with letters, numbers, and `_`.

Import a module

- Import the whole module as a namespace

```
## Import the module mdl
import mdl
## Use a function foo in the module mdl
mdl.foo()
```

- Extract one item (e.g., a function) in the module

```
## Only import the foo function, not the whole module
from mdl import foo
## Use this function; no need to add mdl. in the front
foo()
```

Note: import all function in a module `from mdl import *` is not recommended because this may cause confusion if multiple modules having items with the same name.

Create a module

In a script file or a library module:

- To enable executing the file in the command line via `./some_file.py`, Line 1 of the file `some_file.py` is

```
#!/usr/bin/env python
```

- The first code is a module level docstring

```
'''
This is a script which does something useful.
'''
```

- A version after the docstring

```
__version__ = '00.05.01'
```

- Then, a group of `import` that this module replies on.

- Finally, the code (that defines new functions).

Test code at the end of the script:

```
if __name__ == '__main__'
```

- At the end of a script, we can add the function evaluation after this `if` statement.
- The global variable `__name__` is set to the module name when an `import` is running, and it is set to `__main__` if the main script is running.
- So when this is run as a script, the code block after this `if` statement will run. When this is imported into another script, the code after this `if` statement won't run.

Create a package

- A package must contain a module named `__init__.py`, and this file can be empty. This differentiate a Python library (which is a folder) with any other folders that contain Python files.
- Avoid to create (often nested) packages. A flat list of modules are preferred.
- Install a package: in the command line,

```
pip install pkg
```

5. Write Code Tests

- Test-driven development: write tests (in a new `test_foo` function) before writing the code for the actual function `foo`.

- Keep tests in separate files from the programs (main functions), e.g., `foo.py` vs `test_foo.py`. In this case, the first line in the test file can be

```
import foo
```

Unite test

- A unit test: one line of `assert` to check one situation of the function.

- We can put multiple unit tests in the unit testing framework. In the following example, we build a class `TestRO` for testing, which has several individual test functions inside.

```
import unittest

class TestRO(unittest.TestCase):
    def test_ro_case1(self):
        self.assertCase1( foo(var1) )
    def test_ro_case2(self):
        self.assertCase2( foo(var2) )

## Run all tests
unittest.main()
```

Notes:

- In the output of `unittest.main()`, the first line is the summary, where `.` stands for a pass and `F` for a failure.

Docstring test (doctest)

- Examples can be copied to the docstring using Python prompt `>>>`.
- After defining the function, we can run the examples in the docstring:

```
import doctest
doctest.testmode()
```

Nose package

- A third party package. It finds all functions that contain `'test'` or `'Test'` as a word or followed by `_` or `-`, and run these functions. We can run Nose in command line,

```
$ nosetests file.py
```

- When the filename contain `'test'`, we can even omit the file name in the command line.

Performance tests: `time`

The `time.clock()` function in the `time` module is similar to the `proc.time()` function in R.

```
import time
s1 = time.clock()
block of code
s2 = time.clock()
print(s2 - s1)
```