Programming Tutorial

05 Module

Don't reinvent the wheel

There are many kinds of efforts to reuse in software development

- Architecture Style
  - MultiLayers Style in Mary Shaw's Architecture Style
- Framework
  - SSH Framework(Struts, Spring, Hibernate) in Java EE
- Libary
  - SWING,AWT in Java Libaries
- Design Pattern
  - Strategy Pattern in Design Pattern
- Code Reuse
  - Super Class Inheritance in Object Oriented Programming

Module, Libary, Package

Module

What if you wanted to reuse a number of functions in other programs that you write? As you might have guessed, the answer is modules. A module can be imported by another program to make use of its functionality.

```
# Save this file as mymodule.py
def say_hi():
    print('Hi, this is  mymodule speaking.')

__version__ = '0.1'
```

```
# Save this file as mymodule_demo.py
import mymodule
mymodule.say_hi()
print('Version', mymodule.__version__)
```

```
from mymodule import say_hi, __version__
say_hi()
print('Version', __version__)
```

Libary

Every Language has its standard library to facilicate us to make use of its functionality, such as time, system, math, etc.

```
import time
thisyear=int(time.strftime("%Y", time.localtime()))
print(thisyear)
```

```
import sys
print('The command line arguments are:')
for i in sys.argv:
    print(i)
print('\n\nThe PYTHONPATH is',sys.path,'\n')
```

```
from math import sqrt
print("Square root of 16 is", sqrt(16))
```

Package

Packages are just a convenience to hierarchically organize modules. You will see many instances of this in the standard library.

Packages are just folders of modules with a special __init__.py file that indicates to

Python that this folder is special because it contains Python modules.

```
- <some folder present in the sys.path>/
  - world/
    - __init__.py
    - asia/
      -  __init__.py
      -  india/
        - __init__.py
        - foo.py
    - africa/
      - __init__.py
      - madagascar/
        - __init__.py
        - bar.py
```

Don't repeat yourself

Don’t Repeat Yourself(DRY) is a software development principle, the main aim of which is to reduce repetition of code.

- The biggest benefit of using DRY is maintainability.
- More often than not, DRY code is more readable.

```
System.out.println(“ctrl c”);
System.out.println(“ctrl c”);
System.out.println(“ctrl c”);
System.out.println(“ctrl c”);

//Use loop to avoid repetition
for(i=0;i<4;i++)
  System.out.println(“ctrl c”);
```

```
for (int i = 0; i < 4; i++)
{
   sum1 += array1[i];
}
average1 = sum1/4;

for (int i = 0; i < 4; i++)
{
   sum2 += array2[i];
}
average2 = sum2/4;


/*
Use method abstraction to avoid repetion
*/
int calcAverage (int* Array_of_4)
{
   int sum = 0;
   for (int i = 0; i < 4; i++)
   {
       sum += Array_of_4[i];
   }
   return sum/4;
}

int average1 = calcAverage(array1);
int average2 = calcAverage(array2);
```

```
private Double getTotalSum(List amounts) {
  double totalToPay = 0.00;
  Iterator amountsIterator = amounts.iterator();
  while (amountsIterator.hasNext()) {
    Amount amount = (Amount) amountsIterator.next();
    if (!cancelstatuses.contains(amount.getStatus())) {
      totalToPay += amount.doubleValue();
    }
  }
  return new Double(totalToPay);
```

```java
}

private Double getTotalSumExcludeCancelAmount(List amounts) {
  double totalToPay = 0.00;
  Iterator amountsIterator = amounts.iterator();
  while (amountsIterator.hasNext()) {
    Amount amount = (Amount) amountsIterator.next();
    if (!amount.getIsToCancel()) {
      // Additional condition comparing to the first method.
      if (!cancelstatuses.contains(amount.getStatus())) {
        totalToPay += amount.doubleValue();
      }
    }
  }
  return new Double(totalToPay);
}

// Use data filter to avoid repetition
private Double getTotalSumExcludeCancelAmount(List amounts) {
     //1. filter the list with “!amount.getIsToCancel()” condition
      List newamounts = filter(amounts);
     //2. get sum
     return getTotalSum(newamounts);
}
```