Web Programming
Python Part II.

Shuo Zhang

Review

- Python basics
- Functions

Roadmap today

- Module
- Class
- Input and Output
- Errors and Exceptions
- Brief tour of standard libraries

Module

- A module is a file containing Python definitions and statements.
- These statements are intended to initialize the module. They are executed only the first time the module name is encountered in an import statement.
- Modules can import other modules.

Module

In fibo.py

```
def fib(n):
    a, b = 0, 1
    while b < n:
        print(b, end=' ')
        a, b = b, a+b
    print()
```

```
def fib2(n):
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

Module

- Now enter the Python interpreter and import this module with the following command:

```
import fibo
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo’
```

Module

```
>>> from fibo import fib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
>>> from fibo import *
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Module

- Executing modules as scripts

```
if __name__ == "__main__":
    import sys
    fib(int(sys.argv[1]))
```

Dir()

- The built-in function dir() is used to find out which names a module defines.

```
>>> import fibo
>>> dir(fibo)
```

Module

- Importing * From a Module(tricky)

Packages

- Packages are a way of structuring Python’s module namespace by using “dotted module names”. For example, the module name A.B designates a submodule named B in a package named A.

Package structure

```
my_math/    # Top-level package
        __init__.py  # Initialize the sound package
        fibo/  # Subpackage for file format conversions
                __init__.py
                fib1.py
                fib2.py
Nordlys
```

Package installing

- https://packaging.python.org/installing/

Exercise 4

- Create modules

Class

- **Class:** A user-defined prototype for an object that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.
- **Class Definition Syntax**

```
class ClassName:
    <statement-1>
     . . .
    <statement-N>
```

Class

- Python classes provide all the standard features of Object Oriented Programming:
    - Multiple base classes
    - A derived class can override any methods of its base class or classes
    - A method can call the method of a base class with the same name
    - Objects can contain arbitrary amounts and kinds of data;

Object

- *Objects* are Python's abstraction for data, when you instantiate a class, what you get is called an object.

  （**Instantiation:** The creation of an instance of a class.）

Class VS Object

Create a class:

```
class MyStuff:
    def __init__(self):
        self.tangerine = “tangerine”
    def apple(self):
        print(“I am an apple!”)
```

Instantiate:

```
thing = MyStuff()
thing.apple()
```

Instantiation

- When a class defines an __init__() method, class instantiation automatically invokes __init__() for the newly-created class instance. It is called class constructor or initialization method that Python calls when you create a new instance of this class.

Instantiation

- Of course, the __init__() method may have arguments for greater flexibility. In that case, arguments given to the class instantiation operator are passed on to __init__(). For example,

```
class Employee: #Common base class for all employees
    empCount = 0
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary
        Employee.empCount += 1
```

Creating Instance Objects

- To create instances of a class, you call the class using class name and pass in whatever arguments its __*init*__ method accepts.

  "This would create first object of Employee class"

  ```
  emp1 = Employee("Zara", 2000)
  ```

  "This would create second object of Employee class"

  ```
  emp2 = Employee("Manni", 5000)
  ```

- Accessing Attributes

  ```
  emp1.displayEmployee()
  ```

Class and Instance Variables

```
class Dog:
    # class variable shared by all instances
    kind = 'canine'
    def __init__(self, name):
        # instance variable unique to each instance
        self.name = name
>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.kind # shared by all dogs 'canine'
>>> e.kind # shared by all dogs 'canine'
>>> d.name # unique to d 'Fido'
>>> e.name # unique to e 'Buddy'
```

Inheritance

```
class DerivedClassName(BaseClassName):
    <statement-1>
    . . .
    <statement-N>
```

- The name BaseClassName must be defined in a scope containing the derived class definition. In place of a base class name, other arbitrary expressions are also allowed.
- When the base class is defined in another module:
- **class DerivedClassName**(modname.BaseClassName):

Multiple Inheritance

- Python supports a form of multiple inheritance as well

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    . . .
    <statement-N>
```

Private Variables and Methods

- Private means the attributes are only available for the members of the class not for the outside of the class.
- Not strict, actually Name mangling

Private Variables and Methods

```
class Mapping:
    def __init__(self):
        self.items_list = [] # public
        self.__num = 0 # private
    def __map(self, iterable):
        pass
    @property
    def num(self):
        return self.__num

x = Mapping()
print(x.num)
print(x._Mapping__num)
```

```
class Employee:  # Create an empty employee record
      pass # Fill the fields of the record
```

```
john = Employee()
john.name = 'John Doe'
john.dept = 'computer lab'
john.salary = 1000
```

Decorator

- Decorators allow you to inject or modify code in functions or classes.
- The @ indicates the application of the decorator.

Staticmethod

- Static method: Method belongs to a class, but that doesn't use the object itself at all.

```
class Pizza(object):
    @staticmethod
    def mix_ingredients(x, y):
        return x + y
```

Class method

- Class methods are methods that are not bound to an object, but to a class!

```
class Pizza(object):
    radius = 42
    @classmethod
    def get_radius(cls):
        return cls.radius
```

- Use class to access: Pizza.get_radius()

Abstract method

- An abstract method is a method defined in a base class, but that may not provide any implementation

```
class Pizza(object):
    def get_radius(self):
        raise NotImplementedError
```

Inheritance vs Composition

- Composition is a way of *aggregating* objects together by making some objects attributes of other objects. “has-a”

- *Inheritance* is a way of arranging objects in a hierarchy from the most general to the most specific. “is a”

- http://python-textbok.readthedocs.io/en/1.0/Object_Oriented_Programming.html

Exercise 5

- CardHolder class

Output formatting

- Methods from last lecture:
    - Expression statement
    - Print()
- If want more control over the formatting of your output than simply printing space-separated values:
    - Handle string manually
    - Use formatted string literals

Substitute values into strings

- The **str()** function is meant to return representations of values which are fairly human-readable
- **repr()** is meant to generate representations which can be read by the interpreter
- For objects which don’t have a particular representation for human consumption, str() will return the same value as repr()

Examples

```
s = 'Hello python'
str(s)
```

'Hello python’

```
repr(s)
```

"'Hello python’”

```
str(1/7)
```

'0.14285714285714285'

str.format()

- The brackets and characters within them (called format fields) are replaced with the objects passed into the str.format() method.

```
print('We are the {} who say \
"{}!"'.format('knights', 'Ni'))
```

We are the knights who say "Ni!”

str.format()

- A number in the brackets can be used to refer to the position of the object passed into the str.format() method.

```
print('{0} and {1}'.format('spam', 'eggs'))
```

spam and eggs

```
print('{1} and {0}'.format('spam', 'eggs'))
```

eggs and spam

str.format()

- If keyword arguments are used in the str.format() method, their values are referred to by using the name of the argument.

```
print('This {food} is {adjective}.'.format(\
food='spam',adjective='absolutely horrible'))
```

This spam is absolutely horrible.

Regular expression operations

- Check a string match a pattern or not(Perl style)
- “re” module

Re.match()

- Check whether the beginning of *string* match the regular expression *pattern*, return a corresponding match object
- re.match(pattern, string, flags=0)

```
import re

print(re.match('www', 'www.runoob.com').span())
print(re.match('com', 'www.runoob.com'))
```

(0, 3)

None

Re.search()

- Scan through string looking for the first location where the regular expression pattern produces a match.
- re.search(pattern, string, flags=0)

```
import re
print(re.search('www', 'www.runoob.com').span())
print(re.search('com', 'www.runoob.com').span())
```

(0, 3)

(11, 14)

Re.search()

```
m = re.search('(?<=-)\w+', 'spam-egg')
m.group(0)
'egg'
```

Regular expression operations

- For more methods and patterns, see

https://docs.python.org/3/library/re.html

Exercise 6

- Output formatting

Reading and Writing Files

- File commands
    - open: opens the file
    - close: closes the file
    - read: reads the contents of the file
    - readline: reads just one line of a text file
    - truncate: empties the file(watch out!)
    - write(stuff): writes stuff to the file

Reading and Writing Files

- open() returns a file object, and is most commonly used with two arguments: open(filename, mode)

```
f = open('my_file.txt', 'r')
f.close() or
with open('my_file', 'a') as f:
    f.write('something')
```

- The first argument is a string containing the filename. The second argument is another string containing a few characters describing the way in which the file will be used.

Reading and Writing Files

- *mode*
    - ’r’:only read
    - ’w’: only writing
    - ’a’: opens the file for appending to the end
    - ’r+' opens the file for both reading and writing.
- The *mode* argument is optional; 'r' will be assumed if it’s omitted.

Read()

- To read a file’s contents, call f.read(size), which reads some quantity of data and returns it as a string (in text mode) or bytes object (in binary mode).
- *size* is an optional numeric argument.

```
f.read()
'This is the entire file.\n'
f.read()
””
```

Readline()

- f.readline() reads a single line from the file;
- A newline character (\n) is left at the end of the string, and is only omitted on the last line of the file if the file doesn’t end in a newline.

Readline()

```
f.readline() 'This is the first line of the file.\n'
f.readline() 'Second line of the file\n'
f.readline() ”
for line in f:
    print(line, end='')
```

This is the first line of the file.

Second line of the file

If you want to read all the lines of a file in a list you can also use list(f) or f.readlines().

Write()

- f.write(string) writes the contents of *string* to the file, returning the number of characters written.
- Other types of objects need to be converted – either to a string (in text mode) or a bytes object (in binary mode) – before writing them.

```
value = ('the answer', 42)
s = str(value) # convert the tuple to string
f.write(s)
18
```

JSON

- JOSN: JavaScript Object Notation
- JSON is a syntax for storing and exchanging data
- When you want to save more complex data types like nested lists and dictionaries, parsing and serializing by hand becomes complicated.

JSON

- **json.dumps():** Encode data
- **json.loads():** Decode data

```
>>> import json
>>> json.dumps([1, 'simple', 'list'])
    '[1, "simple", "list"]’
```

Encode

Python	JSON
dict	object
list, tuple	array
str	string
int, float, int- & float-derived Enums	number
True	true
False	false
None	null

Decode

JSON	Python
object	dict
array	list
string	str
number (int)	int
number (real)	float
true	True
false	False
null	None

JSON

- Another variant of the dumps() function, called dump() simply serializes the object to a text file. So if f is a text file object opened for writing, we can do this:

```
json.dump(x, f)
```

- To decode the object again, if f is a text file object which has been opened for reading:

```
x = json.load(f)
```

JSON

```
# Write JSON data into file
with open('data.json', 'w') as f:
     json.dump(data, f)
# Read data from file
with open('data.json', 'r') as f:
     data = json.load(f)
```

Pickle

- pickle - the pickle module
- Contrary to JSON, *pickle* is a protocol which allows the serialization of arbitrarily complex Python objects.
- https://docs.python.org/3/library/pickle.html#data-stream-format

Exercise 7

- Read file
- JSON write

Errors and Exceptions

- Syntax Errors
- Exceptions

Syntax Errors

- Syntax errors, also known as parsing errors, are perhaps the most common kind of complaint you get while you are still learning Python:

```
while True
        print('Hello world')
```

- A colon (':') is missing before it.

Exceptions

- Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called *exceptions* and are not unconditionally fatal

Exceptions

```
10 * (1/0)
```

Traceback (most recent call last): File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero

```
4 + spam*3
```

Traceback (most recent call last): File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined

```
'2' + 2
```

Traceback (most recent call last): File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly

Handling Exceptions

```
while True:
    try:
        x = int(input("Please enter a number: "))
        break
    except ValueError:
        print("Oops! That was no valid number. Try again...")
```

Handling Exceptions

- A *try* statement may have more than one except clause, to specify handlers for different exceptions. At most one handler will be executed.

```
except (RuntimeError, TypeError, NameError):
        pass
```

Handling Exceptions

- The try ... Except statement has an optional *else clause*

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except OSError:
        print('cannot open', arg)
    else:
        print(arg, 'has', len(f.readlines()), 'lines')
f.close()
```

Raising Exceptions

- The *raise* statement allows the programmer to force a specified exception to occur.

```
raise NameError('HiThere')
```

Traceback (most recent call last): File "<stdin>", line 1, in <module>
NameError: HiThere

User-defined Exceptions

- Programs may name their own exceptions by creating a new exception class
- Exceptions should typically be derived from the Exception class, either directly or indirectly

Defining Clean-up Actions

- The *try* statement has another optional clause which is intended to define clean-up actions that must be executed under all circumstances

```
try:
        raise KeyboardInterrupt
finally:
        print('Goodbye!')
```

Goodbye!

KeyboardInterrupt Traceback (most recent call last): File "<stdin>", line 2, in <module>

Exercise 8

- Exception

Brief tour of the standard libraries

1. Operating System: os
2. Command Line Arguments: sys
3. String Pattern Matching: re
4. Mathematics: math; random; statistics
5. Internet Access: urllib.request
6. Machine learning: Tensorflow; Theano; scikit-learn; etc
7. Database: pymysql