HCIA-AI Huawei Certification Course

HCIA-AI
Python Programming Basics
Experimental Guide

Issue: 1.0

HUAWEI

HUAWEI TECHNOLOGIES CO., LTD.

Huawei Technologies Co., Ltd.

Address:	Huawei Industrial Base
	Bantian, Longgang
	Shenzhen 518129
	People's Republic of China
Website:	http://e.huawei.com

Introduction to Huawei Certification System

Based on cutting-edge technologies and professional training systems, Huawei certification meets the diverse AI technology demands of various clients. Huawei is committed to providing practical and professional technical certification for our clients.

HCIA-AI V1.0 certification is intended to popularize AI and help understand deep learning and Huawei Cloud EI, and learn the basic capabilities of programming based on the TensorFlow framework, as a motive to promote talent training in the AI industry.

Content of HCIA-AI V1.0 includes but is not limited to: AI overview, Python programming and experiments, mathematics basics and experiments, TensorFlow introduction and experiments, deep learning pre-knowledge, deep learning overview, Huawei cloud EI overview, and application experiments for image recognition, voice recognition and man-machine dialogue.

HCIA-AI certification will prove that you systematically understand and grasp Python programming, essential mathematics knowledge in AI, basic programming methods of machine learning and deep learning platform TensorFlow, pre-knowledge and overview of deep learning, overview of Huawei cloud EI, basic programming for image recognition, voice recognition, and man-machine dialogue. With this certification, you have required knowledge and techniques for AI pre-sales basic support, AI after-sales technical support, AI products sales, AI project management, and are qualified for positions such as natural language processing (NLP) engineers, image processing engineers, voice processing engineers and machine learning algorithm engineers.

Enterprises with HCIA-AI-certified engineers have the basic understanding of AI technology, framework, and programming, and capable of leveraging AI, machine learning, and deep learning technologies, as well as the open-source TensorFlow framework to design and develop AI products and solutions like machine learning, image recognition, voice recognition, and man-machine dialogue.

Huawei certification will help you open the industry window and the door to changes, standing in the forefront of the AI world!

Preface

Brief Introduction

This document is an HCIA-AI certification training course, intended to trainees who are preparing for HCIA-AI tests or readers who want to know about AI basics. After understanding this document, you can perform simple Python basic programming, laying a solid foundation for later AI development.

Contents

This experimental guide includes nine experiments, covering Python programming basics, and is intended to help trainees and readers easily develop capabilities of developing AI.

- Experiment 1: Understand definition and operations of lists and tuples of Python.
- Experiment 2: Understand definition and operations of Python strings.
- Experiment 3: Understand definition and operations of Python dictionaries.
- Experiment 4: Understand definition and operations of conditional statements and looping statements of Python.
- Experiment 5: Understand definition and operations of Python functions.
- Experiment 6: Understand definition and operations object-oriented programming of Python.
- Experiment 7: Understand date and time operations of Python.
- Experiment 8: Understand definition and operations of regular expressions of Python.
- Experiment 9: Understand definition and operations of Python file manipulation.

Knowledge Background of Readers

This course is intended for Huawei certification. To better understand content of this document, readers are required to meet the following basic conditions:

- Have the basic language editing capability.
- Have basic knowledge of data structures and database.

Experimental Environment

This experimental environment is compiled on Python 3.6.

Contents

1 Lists and Tuples

1.1 Introduction to the Experiment

1.1.1 About the Experiment

This experiment introduces knowledge units about lists and tuples in Python, and related operations on them.

1.1.2 Objectives of the Experiment

- Understand meanings of Python lists.
- Understand meanings of Python tuples.
- Grasp related operations on Python lists.
- Grasp related operations on Python tuples.

1.2 Experimental Tasks

1.2.1 Concepts

- Python lists. Lists include a series of ordered elements, and expressed in the form of []. As lists are combined orderly, you can identify the position of elements for access.

Note: The list index starts from 0 forward and –1 backward.

- Python tuples. Tuples are similar to lists in Python, except for that elements of tuples are unchangeable. Tuples are expressed in parentheses while lists in square brackets. It is easy to create tuples, simply by adding elements in brackets and separating them with commas.

1.2.2 Experimental Operations on Lists

This section describes related operations on lists.

1. Understand the difference between "append" and "extend".

"Append" adds data to the end of a list as a new element. Its arguments can be any object.

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

```
>>>x.append(y)
>>>print(x)
[1, 2, 3, [4, 5]]
```

The argument of "extend" must be an iterated object, which means that all elements of this object are added to the end of list one by one.

```
>>>x = [1, 2, 3]
>>>y = [4, 5]
>>>x.extend(y)
>>>print(x)
[1, 2, 3, 4, 5]
# equal to
>>>for i in y:
>>>x.append(i)
>>>print(x)
```

2. Check whether the list is empty.

```
if len(items) == 0:
print("empty list")
or
if items == []:
print("empty list")
```

3. Copy a list.

```
Method one:
new_list = old_list[:]
Method two:
new_list = list(old_list)
Method three:
import copy
new_list = copy.copy(old_list)# copy
new_list = copy.deepcopy(old_list)# deep copy
```

4. Get the last element of a list.

Elements in an index list can be positive or negative numbers. Positive numbers mean indexing from the left of list, while negative numbers mean indexing from the right of list. There are two methods to get the last element.

```
>>> a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> a[len(a)-1]
10
>>> a[-1]
10
```

5. Sort lists.

You can use two ways to sort lists. Python lists have a sort method and a built-in function (sorted). You can sort complicated data types by specifying key arguments. You can sort lists composed of dictionaries according to the age fields of elements in the dictionaries.

```
>>>items = [{'name': 'Homer', 'age': 39},
{'name': 'Bart', 'age': 10},
{"name": 'cater', 'age': 20}]
>>>items.sort(key=lambda item: item.get("age"))
>>>print(items)
```

```
[{'age': 10, 'name': 'Bart'}, {'age': 20, 'name': 'cater'}, {'age': 39, 'name': 'Homer'}]
```

6. Remove elements from a list.

The "remove" method removes an element, and it removes only the element that appears for the first time.

```
>>> a = [0, 2, 2, 3]
>>> a.remove(2)
>>> a
[0, 2, 3]
# ValueError will be returned if the removed element is not within the list.
>>> a.remove(7)
Traceback (most recent call last):
   File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list·
```

"del" removes a certain element in a specified position.

"pop" is similar to "del", but "pop" can return the removed element.

7. Connect two lists.

```
>>>listone = [1, 2, 3]
>>>listtwo = [4, 5, 6]
>>>mergedlist = listone + listtwo
>>>print(mergelist)
[1, 2, 3, 4, 5, 6]
```

1.2.3 Experimental Operations on Tuples

This section describes related operations on tuples.

Another type of ordered lists is called tuples and is expressed in (). Similar to a list, a tuple cannot be changed after being initialized. Elements must be determined when a tuple is defined.

A tuple has no append() and insert() methods and cannot be assigned as another element. The method for getting a tuple is similar to that for getting a list.

As tuples are unchangeable, code is more secure. Therefore, if possible, use tuples instead of lists.

1. Define a tuple for an element.

```
t=(1,)
```

Note: In t=(1), t is not a tuple type, as the parentheses () can represent a tuple, or mathematical formula. Python stipulates that in this case, () is a mathematical formula, and a tuple with only one element must have a comma to eliminate ambiguity.

2. Define a changeable tuple.

```
>>> cn=('yi','er','san')
>>> en=('one','two','three')
>>> num=(1,2,3)
>>> tmp=[cn,en,num,[1.1,2.2],'language']
>>>print(tmp)
[('yi', 'er', 'san'), ('one', 'two', 'three'), (1, 2, 3), [1.1, 2.2], 'language']
>>>print(tmp[0])
('yi', 'er', 'san')
```

```
>>>print(tmp[0][0])
yi
>>>print(tmp[0][0][0])
y
```

2 Strings

2.1 Introduction to the Experiment

2.1.1 About the Experiment

This experiment mainly introduces related knowledge units about strings in Python, and related operations on them.

2.1.2 Objectives of the Experiment

- Understand meanings of Python strings.
- Grasp related operations on Python strings.

2.2 Experimental Tasks

2.2.1 Concepts

- Strings of Python: A string is a sequence composed of zero or multiple characters, and it is one of the six built-in sequences of Python. Strings are unchangeable in Python, which are string constants in C and C++ languages.
- Expression of strings. Strings may be expressed in single quotes, double quotes, triple quotes, or as escape characters and original strings.

2.2.2 Experimental Operations

1. Single quotes and double quotes

Strings in single quotes are equal to those in double quotes, and they are exchangeable.

```
>>> s = 'python string'
>>>print(s)
python string
>>> ss="python string"
>>>print(ss)
python string
>>> sss='python "Hello World"string'
>>>print(sss)
```

```
python "Hello World"string
```

2. Long strings

Triple quotes can define long strings in Python as mentioned before. Long strings may have output like:

```
>>>print('''this is a long string''')
this is a long string
```

3. Original strings

Original strings start with r, and you can input any character in original strings. The output strings include backslash used by transference at last. However, you cannot input backslash at the end of strings. For example:

```
>>> rawStr = r'D:\SVN_CODE\V900R17C00_TRP\omu\src'
>>>print(rawStr)
D:\SVN_CODE\V900R17C00_TRP\omu\src
```

4. Width, precision, and alignment of strings

To achieve the expected effects of strings in aspects of width, precision, and alignment, refer to the formatting operator commands.

```
>>>print("%c" % 97)
a
>>>print("%6.3f" % 2.5)
  2.500
>>>print("%+10x" % 10)
                +a
>>>print("%.*f" % (4, 1.5))
1.5000
```

5. Connect and repeat strings

In Python, you can use "+" to connect strings and use "*" to repeat strings.

```
>>> s = 'I' + 'want' + 'Python' + '.'
>>>print(s)
IwantPython.
>>> ss='Python'*3
>>>print(ss)
PythonPythonPython
```

6. Delete strings

You can use "del" to delete a string. After being deleted, this object will no longer exist, and an error is reported when you access this object again.

```
>>> ss='Python'*3
>>>print(ss)
PythonPythonPython
>>> del ss
>>>print(ss)
Traceback (most recent call last):
   File "<pyshell#35>", line 1, in <module>
print(ssNameError: name 'ss' is not defined)
```

3 Dictionaries

3.1 Introduction to the Experiment

3.1.1 About the Experiment

This experiment mainly introduces related knowledge units about dictionaries in Python, and related operations on them.

3.1.2 Objectives of the Experiment

- Understand meanings of Python dictionaries.
- Grasp related operations on Python dictionaries.

3.2 Experimental Tasks

3.2.1 Concepts

- Python dictionary. A dictionary has a data structure similar to a mobile phone list, which lists names and their associated information. In a dictionary, the name is called a "key", and its associated information is called "value". A dictionary is a combination of keys and values.
- Its basic format is as follows:

```
d = {key1 : value1, key2 : value2 }
```

- You can separate a key and a value with a colon, separate each key/value pair with a comma, and include the dictionary in a brace.
- Some notes about keys in a dictionary: Keys must be unique, and must be simple objects like strings, integers, floating numbers, and bool values.

3.2.2 Experimental Operations

1. Create a dictionary

A dictionary can be created in multiple manners, as shown below.

```
>>> a = {'one': 1, 'two': 2, 'three': 3}
>>>print(a)
```

```
{'three': 3, 'two': 2, 'one': 1}
>>> b = dict(one=1, two=2, three=3)
>>>print(b)
{'three': 3, 'two': 2, 'one': 1}
>>> c = dict([('one', 1), ('two', 2), ('three', 3)])
>>>print(c)
{'three': 3, 'two': 2, 'one': 1}
>>> d = dict(zip(['one', 'two', 'three'], [1, 2, 3]))
>>>print(d)
{'three': 3, 'two': 2, 'one': 1}
>>> e = dict({'one': 1, 'two': 2, 'three': 3})
>>>print(e)
{'one': 1, 'three': 3, 'two': 2}
>>>print(a==b==c==d==e)
True
```

2. dictcomp

"dictcomp" can build a dictionary from iterated objects that use key/value pairs as elements.

```
>>> data = [("John","CEO"),("Nacy","hr"),("LiLei","engineer")]
>>> employee = {name:work for name, work in data}
>>>print(employee)
{'LiLei': 'engineer', 'John': 'CEO', 'Nacy': 'hr'}
```

3. Dictionary lookup

Look up directly according to a key value.

```
>>>print(employee["John"])
CEO
```

If there is no matched key value in a dictionary, KeyError is returned.

```
>>>print(employee["Joh"])
Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
print(employee["Joh"])
KeyError: 'Joh'
```

When you use dic[key] to look up a key value in a dictionary, it will return an error if there is no such key value. However, if you use dic.get(key, default) to look up a key value, it will return default if there is no such key value.

```
>>>print(employee.get("Nacy","UnKnown'"))
hr
>>>print(employee.get("Nac","UnKnown"))
UnKnown
```

4 Conditional and Looping Statements

4.1 Introduction to the Experiment

4.1.1 About the Experiment

This experiment mainly introduces related knowledge and operations about conditional and looping statements in Python.

4.1.2 Objectives of the Experiment

- Understand meanings of conditional and looping statements in Python.
- Understand related operations on conditional and looping statements in Python.

4.2 Experimental Tasks

4.2.1 Concepts

1. In Python, the "if" statement controls execution of a program, and its basic form is as follows:

```
if judging condition:
    Execute statement…
else:
    Execute statement …
```

The following statements are executed if the judging condition is true (non-zero). There can be multiple lines of execution statements, which can be indented to indicate the same range. The "else" statement is optional, and can be executed when the condition is false.

2. Looping statement

There are a lot of changes in looping statements. Common statements include the "for" statement and the "while" statement.

In "for" looping, the "for" statement should be followed by a colon. "for" looping is performed in a way similar to iterating. In "while" looping, there is a judgment on condition and then looping, like in other languages.

4.2.2 Experimental Operations

1. "for" looping

```
>>> for i in range(0,10):
>>>print(i)
0
1
2
3
4
5
6
7
8
9
>>> a=[1,3,5,7,9]
>>> for i in a:
>>>print(i)
1
3
5
7
9
```

2. "while" looping

```
>>>while (a>100):
>>>print(a)
[1, 3, 5, 7, 9]
[1, 3, 5, 7, 9]
[1, 3, 5, 7, 9]
[1, 3, 5, 7, 9]
[1, 3, 5, 7, 9]
[1, 3, 5, 7, 9]
[1, 3, 5, 7, 9]
[1, 3, 5, 7, 9]
[1, 3, 5, 7, 9]
[1, 3, 5, 7, 9]
[1, 3, 5, 7, 9]
[1, 3, 5, 7, 9]
[1, 3, 5, 7, 9]
[1, 3, 5, 7, 9]
[1, 3, 5, 7, 9]
[1, 3, 5, 7, 9]
[1, 3, 5, 7, 9]
[1, 3, 5, 7, 9]
[1, 3, 5, 7, 9]
[1, 3, 5, 7, 9]
[1, 3, 5, 7, 9]
[1, 3, 5, 7, 9]
[1, 3, 5, 7, 9]
```

5 Functions

5.1 Introduction to the Experiment

5.1.1 About the Experiment

This experiment mainly introduces related knowledge and operations about functions in Python.

5.1.2 Objectives of the Experiment

- Understand meanings of Python functions.
- Grasp related operations on Python function.

5.2 Experimental Tasks

5.2.1 Concepts

Functions can raise modularity of applications and reuse of code. In Python, strings, tuples, and numbers are unchangeable, while lists and dictionaries are changeable. For those unchangeable types such as integers, strings, and tuples, only values are transferred during function calling, without any impact on the objects themselves. For those changeable types, objects are transferred during function calling, and external objects will also be impacted after changes.

5.2.2 Experimental Operations

1. Common built-in functions

The "int" function can be used to convert other types of data into integers.

```
>>> int('123')
123
>>> int(12.34)
12
>>> float('12.34')
12.34
>>> str(1.23)
```

```
'1.23'
>>> str(100)
'100'
>>> bool(1)
True
>>> bool('')
False
```

2. Function name

A function name is a reference to a function object, and it can be assigned to a variable, which is equivalent to giving the function an alias.

```
>>> a = abs # Variable a points to function abs
>>> a(-1) # Therefore, the "abs" can be called by using "a"
1
```

3. Define functions

In Python, you can use the "def" statement to define a function, listing function names, brackets, arguments in brackets and colons successively. Then you can edit a function in an indentation block and use the "return" statement to return values.

We make an example by defining the "my_abs" function to get an absolute value.

```
>>> def my_abs(x):
        if x>=0:
            return x
        else:
            return –x
```

You can use the "pass" statement to define a void function, which can be used as a placeholder. Change the definition of "my_abs" to check argument types, that is, to allow only arguments of integers and floating numbers. You can check data types with the built-in function isinstance().

```
def my_abs(x):
    if not isinstance(x, (int, float)):
        raise TypeError('bad operand type')
    if x >= 0:
        return x
    else:
        return –x
```

4. Keyword arguments

Changeable arguments allow you input zero or any number of arguments, and these changeable arguments will be assembled into a tuple for function calling. While keyword arguments allow you to input zero or any number of arguments, and these keyword arguments will be assembled into a dictionary in functions.

```
def person(name, age, **kw):
print('name:', name, 'age:', age, 'other:', kw)
```

Function "person" receives keyword argument "kw" except essential arguments "name" and "age". You can only input essential arguments when calling this function.

```
>>> person('Michael', 30)
name: Michael age: 30 other: {}
```

You can also input any number of keyword arguments.

```
>>> person('Bob', 35, city='Beijing')
name: Bob age: 35 other: {'city': 'Beijing'}
>>> person('Adam', 45, gender='M', job='Engineer')
name: Adam age: 45 other: {'gender': 'M', 'job': 'Engineer'}
```

Similar to changeable arguments, you can assemble a dictionary and convert it into keyword arguments as inputs.

```
>>> extra = {'city': 'Beijing', 'job': 'Engineer'}
>>> person('Jack', 24, city=extra['city'], job=extra['job'])
name: Jack age: 24 other: {'city': 'Beijing', 'job': 'Engineer'}
```

You can certainly simplify the above-mentioned complex function calling.

```
>>> extra = {'city': 'Beijing', 'job': 'Engineer'}
>>> person('Jack', 24, **extra)
name: Jack age: 24 other: {'city': 'Beijing', 'job': 'Engineer'}
```

**extra means transferring all key-values in this extra dictionary as key arguments into the **kw argument of the function. kw will get a dictionary, which is a copy of the extra dictionary. Changes on kw will not impact the extra dictionary outside the function.

5. Name keyword arguments

If you want to restrict names of keyword arguments, you can name keyword arguments. For example, you can accept only "city" and "job" as keyword arguments. A function defined in this way is as follows:

```
def person(name, age, *, city, job):
print(name, age, city, job)
```

Different from keyword argument "**kw", a special separator "*" is required to name a keyword argument. Arguments after "*" are regarded as naming keyword arguments, which are called as follows:

```
>>> person('Jack', 24, city='Beijing', job='Engineer')
Jack 24 Beijing Engineer
```

The special separator "*" is not required in the keyword argument after a changeable argument in a function.

```
def person(name, age, *args, city, job):
print(name, age, args, city, job)
```

You need to input an argument name to name a keyword argument, which is different from the position argument. An error will be returned during calling if no argument name is introduced. In this case, the keyword argument can be default, as one way to simplify calling.

```
def person(name, age, *, city='Beijing', job):
print(name, age, city, job)
```

Because the keyword argument "city" has a default value, you do not need to input a parameter of "city" for calling.

```
>>> person('Jack', 24, job='Engineer')
Jack 24 Beijing Engineer
```

When you name a keyword argument, "*" must be added as a special separator if there are no changeable arguments. Python interpreter cannot identify position arguments and keyword arguments if there is no "*".

6. Argument combination

To define a function in Python, you can use required arguments, default arguments, changeable arguments, keyword arguments and named keyword arguments. These five types of arguments can be combined with each other.

Note: Arguments must be defined in the order of required arguments, default arguments, changeable arguments, named keyword arguments, and keyword arguments.

For example, to define a function that includes the above-mentioned arguments:

```
def f1(a, b, c=0, *args, **kw):
print('a =', a, 'b =', b, 'c =', c, 'args =', args, 'kw =', kw)
def f2(a, b, c=0, *, d, **kw):
print('a =', a, 'b =', b, 'c =', c, 'd =', d, 'kw =', kw)
```

Python interpreter will input the matched arguments according to argument positions and names automatically when calling a function.

```
>>> f1(1, 2)
a = 1 b = 2 c = 0 args = () kw = {}
>>> f1(1, 2, c=3)
a = 1 b = 2 c = 3 args = () kw = {}
>>> f1(1, 2, 3, 'a', 'b')
a = 1 b = 2 c = 3 args = ('a', 'b') kw = {}
>>> f1(1, 2, 3, 'a', 'b', x=99)
a = 1 b = 2 c = 3 args = ('a', 'b') kw = {'x': 99}
>>> f2(1, 2, d=99, ext=None)
a = 1 b = 2 c = 0 d = 99 kw = {'ext': None}
```

The most amazing thing is that you can call the above-mentioned function through a tuple and a dictionary.

```
>>> args = (1, 2, 3, 4)
>>> kw = {'d': 99, 'x': '#'}
>>> f1(*args, **kw)
a = 1 b = 2 c = 3 args = (4,) kw = {'d': 99, 'x': '#'}
>>> args = (1, 2, 3)
>>> kw = {'d': 88, 'x': '#'}
>>> f2(*args, **kw)
a = 1 b = 2 c = 3 d = 88 kw = {'x': '#'}
```

Therefore, you can call a function through the forms similar to "func(*args, **kw)", no matter how its arguments are defined.

7. Recursive function

You need to prevent a stack overflow when you use a recursive function. Functions are called through the stack which is a data structure in computers. The stack will add a layer of stack frames when a function is called, while the stack will remove a layer of stack frames when a function is returned. As the size of a stack is limited, it will lead to a stack overflow if there are excessive numbers of recursive calling of functions.

Solution to a stack overflow: tail recursion optimization

You can use tail recursion optimization to solve a stack flow. As tail recursion enjoys the same effects with looping, you can take looping as a special tail recursion, which means to call itself when the function is returned and exclude expressions in the "return" statement. In this way, the compiler or interpreter can optimize tail recursion, making recursion occupying only one stack frame, no matter how many times the function is called. This eliminates the possibility of stack overflow.

For the fact(n) function, because a multiplication expression is introduced in return n * fact(n - 1), it is not tail recursion. To change it into tail recursion, more code is needed to transfer the product of each step into a recursive function.

```
def fact(n):
    return fact_iter(n, 1)
def fact_iter(num, product):
    if num == 1:
        return product
    return fact_iter(num - 1, num * product)
```

It can be learned that return fact_iter(num - 1, num * product) returns only the recursive function itself. num – 1 and num * product will be calculated before the function is called, without any impact on the function.

6 Object-Oriented Programming

6.1 Introduction to the Experiment

6.1.1 About the Experiment

This experiment mainly introduces related knowledge and operations about object-oriented programming in Python.

6.1.2 Objectives of the Experiment

- Understand the meaning concerning object-oriented programming in Python.
- Grasp related operations concerning object-oriented programming in Python.

6.2 Experimental Tasks

6.2.1 Concepts

- Object-oriented programming

  As a programming idea, Object Oriented Programming (OOP) takes objects as the basic units of a program. An object includes data and functions that operate the data.

  Process-oriented design (OOD) takes a program as a series of commands, which are a group of functions to be executed in order. To simplify program design, OOD cuts functions further into sub-functions. This reduces system complexity by cutting functions into sub-functions.

  OOP takes a program as a combination of objects, each of which can receive messages from other objects and process these messages. Execution of a computer program is to transfer a series of messages among different objects.

  In Python, all data types can be regarded as objects, and you can customize objects. The customized object data types are classes in object-orientation.
- Introduction to the object-oriented technology
  - Class: A class refers to the combination of objects that have the same attributes and methods. It defines the common attributes and methods of these objects in the combination. Objects are instances of classes.

- Class variable: Class variables are publicly used in the total instantiation, and they are defined within classes but beyond function bodies. Class variables are not used as instance variables.
- Data member: Class variables or instance variables process data related to classes and their instance objects.
- Method re-writing: If the methods inherited from parent classes do not meet the requirements of sub-classes, the methods can be re-written. Re-writing a method is also called overriding.
- Instance variable: Instance variables are defined in methods and are used only for the classes of current instances.
- Inheritance: Inheritance means that a derived class inherits the fields and methods from a base class, and it allows taking the objects of derived class as the objects of base classes. For example, a dog-class object drives from an animal-class object. This simulates a "(is-a)" relationship (in the figure, a dog is an animal).
- Instantiation: It refers to creating instances for a class or objects for a class.
- Methods: functions defined in classes.
- Objects: data structure objects defined through classes. Objects include two data members (class variable and instance variable), and methods.

6.2.2 Experimental Operations

1. Create and use a class

Create a dog class.

Each instance created based on a dog class stores name and age. We will assign capabilities of sitting (sit ()) and rolling over (roll_over ()) as follows:

```
class Dog():
    """a simple try of simulating a dog"""
    def init (self,name,age):
        """Initializeattribute: name and age"""
        self.name = name
        self.age = age
        def sit(self):
            """Simulate sitting when a dog is ordered to do so"""
            print(self.name.title()+"is now sitting")
            def roll_over(self):
                """Simulate rolling over when a dog is ordered to do so"""
                print(self.name.title()+"rolled over!")
```

2. Access attributes

Let us see a complete instance.

```
class Employee:
'All employees base class'
   empCount = 0
   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1
   def displayCount(self):
print("Total Employee %d" % Employee.empCount )
   def displayEmployee(self):
```

```
print("Name : ", self.name,    ", Salary: ", self.salary)
"Create the first object of the employee class"
emp1 = Employee("Zara", 2000)
"Create the second object of the employee class"
emp2 = Employee("Manni", 5000)
emp1.displayEmployee()
emp2.displayEmployee()
print("Total Employee %d" % Employee.empCount)
```

Output following execution of the above-mentioned code:

```
Name :    Zara ,Salary:    2000
Name :    Manni ,Salary:    5000
Total Employee 2
```

3. Class inheritance

The major benefit of oriented-object programming is reuse of code. One way to reuse code is the inheritance mechanism. Inheritance can be taken as setting relationships of parent classes and child classes between classes.

Some features of class inheritance in Python.

- The construction (_init_() method) of the base class will be not auto-called, and it has to be specially called in the construction of its derived classes.
- Class prefixes and self argument variables have to be added to the base class when its methods are called. The self argument is not required when regular functions in classes are called.
- Python always loops up the methods of the corresponding classes, and checks the base class one method by one method only if the methods are not found in the derived classes. (That is, Python searches for the calling method in this class first and then in the base class).
- If an inheritance type lists more than one class, this inheritance is called multi-inheritance.

```
class Parent:              # Define the parent class
    parentAttr = 100
    def __init__(self):
print("Call parent class construction method")
    def parentMethod(self):
print('Call parent class method')
    def setAttr(self, attr):
Parent.parentAttr = attr
    def getAttr(self):
print("Parent class attribute", Parent.parentAttr)
class Child(Parent): # Define a sub-class
    def __init__(self):
print("Call sub-class construction method”)
    def childMethod(self):
print(‘Call sub-class method‘)
c = Child()               # Instantiate sub-class
c.childMethod()           # Call sub-class method
c.parentMethod()          # Call parent class method
c.setAttr(200)            # Re-call parent class method - set attributes
c.getAttr()               # Re-call parent class method - get attributes
```

The code will be executed to output the following results:

Calling the child class construction method;

Calling the child class method;

Calling the parent class method;

Parent class attribute: 200.

4. Class attributes and methods

- Private attributes of classes:

  __private_attrs: It starts with two underlines to indicate a private attribute, which cannot be used outside a class or directly accessed. When it is used inside a class method, follow the form of self.__private_attrs.

- Method of class

  Inside a class, the def keyword can be used to define a method; unlike a regular function, a class method must include the self argument, which has to be the first argument.

- Private method

  __private_method: It starts with two underlines to indicate a private method, which cannot be used outside a class. When it is used inside a class, follow the form of self.__private_methods.

```
class JustCounter:
    __secretCount = 0    # Private variable
    publicCount = 0       # Public variable
    def count(self):
        self.__secretCount += 1
        self.publicCount += 1
print(self.__secretCount)
counter = JustCounter()
counter.count()
counter.count()
print(counter.publicCount)
print(counter.__secretCount)    # Error. Instance cannot access private variable
```

7 Date and Time

7.1 Introduction to the Experiment

7.1.1 About the Experiment

This experiment mainly introduces related knowledge and operations about date and time in Python.

7.1.2 Objectives of the Experiment

- Understand the meaning of date and time in Python.
- Grasp the basic operations on data and time in Python.

7.2 Experimental Tasks

7.2.1 Concepts

How to process date and time is a typical problem for Python. Python provides the time and calendar modules to format date and time.

Time spacing is floating numbers with seconds as the unit. Each time stamp is expressed as the time that has passed since the midnight of January 1st 1970.

The time module of Python has many functions to convert common date formats.

7.2.2 Experimental Operations

1. Get the current time

```
>>>import time
>>>localtime = time.localtime(time.time())
>>>print("Local time:”, localtime)
Output:
Local time: time.struct_time(tm_year=2018, tm_mon=4, tm_mday=28, tm_hour=10, tm_min=3, tm_sec=27,
tm_wday=3, tm_yday=98, tm_isdst=0)
```

2. Get the formatted time

You can choose various formats as required, but the simplest function to get the readable time mode is asctime():

```
>>>import time
>>>localtime = time.asctime( time.localtime(time.time()) )
>>>print("Local time :", localtime)
```

Output:

```
Local time: Thu Apr    7 10:05:21 2016
```

3. Format date

```
>>>import time
# Format into 2016-03-20 11:45:39
>>>print(time.strftime("%Y-%m-%d %H:%M:%S", time.localtime()))
# Format into Sat Mar 28 22:24:24 2016
>>>print(time.strftime("%a %b %d %H:%M:%S %Y", time.localtime()))
# Turn format string into timestamp
>>>a = "Sat Mar 28 22:24:24 2016"
>>>print(time.mktime(time.strptime(a,"%a %b %d %H:%M:%S %Y")))
```

Output:

```
2016-04-07 10:25:09
Thu Apr 07 10:25:09 2016
1459175064.0
```

4. Get calendar of a month

The calendar module can process yearly calendars and monthly calendars using multiple methods, for example, printing a monthly calendar.

```
>>>import calendar

>>>cal = calendar.month(2016, 1)
>>>print("output calendar of January 2016:”)
>>>print(cal)
Output:
The following is the calendar of April 2014
     April 2014
Mo Tu We Th Fr Sa Su
     1
2 3 4 5 6 7 8
9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30
```

8 Regular Expressions

8.1 Introduction to the Experiment

8.1.1 About the Experiment

This experiment mainly introduces related knowledge and operations about regular expressions in Python.

8.1.2 Objectives of the Experiment

- Understand the meaning of regular expressions in Python.
- Grasp the basic operations on regular expressions in Python.

8.2 Experimental Tasks

8.2.1 Concepts

Python regular expressions are special sequences of characters and they enable easy check on whether a string matches a mode.

Python of version 1.5 and later has the new re module, which provides a Perl-style regular expression mode.

The re module enables Python to have all regular expression functions.

The compile function creates a regular expression object based on a mode string and optional flag arguments, and this object has a series of methods for matching and replacing regular expressions.

The re module provides functions that have identical functions to these methods, and these functions use a mode string as the first argument.

This chapter introduces the common Python functions that process regular expressions.

8.2.2 Experimental Operations

1. re.match function

re.match tries to match a mode from the string start position. If no mode is matched from the string start, match() returns none.

Function syntax:

```
re.match(pattern, string, flags=0)
Instance:
>>>import re
>>>print(re.match('www', 'www.runoob.com').span())    # Match at start
>>>print(re.match('com', 'www.runoob.com'))                    # Match not at start
```

Output:

```
(0, 3)
None
```

2. re.search method

re.search scans the entire string and returns the first successful match.

Function syntax:

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

Instance:

```
>>>import re
>>>line = "Cats are smarter than dogs"
>>>searchObj = re.search( r'(.*) are (.*?) .*', line, re.M|re.I)
>>>if searchObj:
>>>print("searchObj.group() : ", searchObj.group())
>>>print("searchObj.group(1) : ", searchObj.group(1))
>>>print("searchObj.group(2) : ", searchObj.group(2))
>>>else:
>>>print("Nothing found!!")
```

Execution result of the above instance:

```
searchObj.group() :    Cats are smarter than dogs
searchObj.group(1) :    Cats
searchObj.group(2) :    smarter
```

3. Differences between re.match and re.search

re.match only matches the string start. If the string start does not agree with the regular expression, the matching fails and the function returns none. re.search matches the entire string until finding a match.

```
>>>import re
>>>line = "Cats are smarter than dogs";
>>>matchObj = re.match( r'dogs', line, re.M|re.I)
>>>if matchObj:
>>>print("match --> matchObj.group() : ", matchObj.group())
>>>else:
>>>print("No match!!")
>>>matchObj = re.search( r'dogs', line, re.M|re.I)
>>>if matchObj:
>>>print("search --> matchObj.group() : ", matchObj.group())
>>>else:
>>>print("No match!!")
```

Execution result of the above instance:

```
No match!!
search --> matchObj.group() :    dogs
```

4. Index and replace

The re module of Python provides re.sub to replace matched items in strings.

Function syntax:

```
re.sub(pattern, repl, string, count=0, flags=0)

>>>import re
>>>phone = "2004-959-559 # This is an oversea telephone number"
# Delete Python comments in strings
>>>num = re.sub(r'#.*$', "", phone)
>>>print("The telephone number is", num)
# Delete non-number (-) strings
>>>num = re.sub(r'\D', "", phone)
>>>print("The telephone number is ", num)
```

Result:

```
The telephone number is: 2004-959-559
The telephone number is:    2004959559
```

5. re.compile function

The compile function compiles regular expressions and creates a regular expression (pattern) object, which will be used by the match() and search() functions.

Function syntax:

```
re.compile(pattern[, flags])

>>>import re
>>> pattern = re.compile(r'\d+')                          # Match at least one number
>>> m = pattern.match('one12twothree34four')              # Search head, no match
>>>print(m)
None
>>> m = pattern.match('one12twothree34four', 2, 10) # Match from ‘e’, no match
>>>print(m)
None
>>> m = pattern.match('one12twothree34four', 3, 10) # Match from ‘1’, matched
>>>print(m)                                                  # Return a match object
<_sre.SRE_Match object at 0x10a42aac0>
>>> m.group(0)     # Ignorable 0
'12'
>>> m.start(0)     # Ignorable 0
3
>>> m.end(0)       # Ignorable 0
5
>>> m.span(0)      # Ignorable 0
(3, 5)
```

6. findall

findall finds all strings that match regular expressions and returns a list. If there is no match, it returns an empty list.

Note: match and search match once, while findall matches all.

Function syntax:

```
findall(string[, pos[, endpos]])

>>>import re
>>>pattern = re.compile(r'\d+')      # Search numbers
>>>result1 = pattern.findall('runoob 123 google 456')
>>>result2 = pattern.findall('run88oob123google456', 0, 10)
>>>print(result1)
>>>print(result2)
```

Output

```
['123', '456']
['88', '12']
```

7. re.finditer

Similar to findall, re.finditer finds all strings that match regular expressions, and returns them as an iterator.

```
re.finditer(pattern, string, flags=0)

>>>import re
>>>it = re.finditer(r"\d+","12a32bc43jf3")
>>>for match in it:
>>>print(match.group())
Output:
12
32
43
3
```

8. re.split

The split method splits matched strings and returns a list. For example:

```
re.split(pattern, string[, maxsplit=0, flags=0])
```

Instance:

```
>>>import re
>>> re.split('\W+', 'runoob, runoob, runoob.')
['runoob', 'runoob', 'runoob', '']
>>> re.split('(\W+)', ' runoob, runoob, runoob.')
['', ' ', 'runoob', ', ', 'runoob', ', ', 'runoob', '.', '']
>>> re.split('\W+', ' runoob, runoob, runoob.', 1)
['', 'runoob, runoob, runoob.']
>>> re.split('a*', 'hello world')      # Split will not split unmatched strings
['hello world']
```

9 File Manipulation

9.1 Introduction to the Experiment

9.1.1 About This Experiment

This experiment mainly introduces related knowledge and operations about file manipulation in Python.

9.1.2 Objectives of the Experiment

- Understand the meaning of file manipulation in Python.
- Grasp the basic operations on file manipulation in Python.

9.2 Experimental Tasks

9.2.1 Concepts

File manipulation is essential to programming languages, as information technologies would be meaningless if data cannot be stored permanently. This chapter introduces common file manipulation in Python.

9.2.2 Experimental Operations

1. Read keyboard input

Python provides two build-in functions to read a text line from the standard input, which a keyboard by default. The two functions are raw_input and input.

raw_input( ) function:

```
>>>str = raw_input("Please input:")
>>>print("Your input is:", str)
```

This reminds your input of any strings and displays the same strings on the screen. When I enter "Hello Python!", it outputs:

```
Please input: Hello Python!
Your input is: Hello Python!
```

input( ) function:

The input([prompt]) and raw_input([prompt]) functions are similar, but the former can receive a Python expression as the input and return the result.

```
>>>str = input("Please input:")
>>>print("Your input is: ", str)
Output:
Please input:[x*5 for x in range(2,10,2)]
Your input is: [10, 20, 30, 40]
```

2. Open and close files

Python provides essential functions and methods to manipulate files by default. You can use the file object to do most file manipulations.

Open() function: You should open a file using the Python build-in open() function, and create a file object, so that the related methods can call it to write and read.

```
# Open a file
>>>fo = open("foo.txt", "w")
>>>print("File name: ", fo.name)
>>>print("closed or not: ", fo.closed)
>>>print("access mode:", fo.mode)
>>>print("space required at head and tail, or not: ", fo.softspace)
```

Output:

```
File name: foo.txt
Close or not:    False
Access mode:    w
Space required at head and tail, or not:    0
```

Close() function: For the file object, it refreshes any buffered data that has not been written and closes the file. Then, the file cannot be written.

When a reference to a file object is re-assigned to another file, Python will close the previous file. It is ideal to close a file using the close() function.

```
# Open a file
>>>fo = open("foo.txt", "w")
>>>print("File name: ", fo.name)
# Close the opened file
>>>fo.close()
```

Output:

File name: foo.txt

3. Write a file

write() function: It writes any string into an opened file. Note that Python strings can be binary data, not just texts. This function will not add a line feed ('\n') at string ends.

```
# Open a file
>>>fo = open("foo.txt", "w")
>>>fo.write( "www.baidu.com!\nVery good site!\n")
# Close an opened file
>>>fo.close()
```

The function above creates a foo.txt file, writes the received content into this file, and closes the file. If you open this file, you will see:

```
www.baidu.com!
Very good site!
```

4. Read a file

Read() function: It reads strings from an opened file. Note that Python strings can be binary data, not just texts.

```
# Open a file
>>>fo = open("foo.txt", "r+")
>>>str = fo.read(10)
>>>print("The read string is: ", str)
# Close an opened file
>>>fo.close()
```

Output:

The read string is: www.runoob.

5. Rename a file

The os module of Python provides a method to execute file processing operations, like renaming and deleting files. To use this module, you have to import it first and then call various functions.

rename(): It requires two arguments: current file name and new file name.

Function syntax:

```
os.rename(current_file_name, new_file_name)

>>>import os
# Rename file test1.txt to test2.txt
>>>os.rename( "test1.txt", "test2.txt" )
```

6. Delete a file

You can use the remove() method to delete a file, using the name of file to be deleted as an argument.

Function syntax:

```
os.remove(file_name)

>>>import os
# Delete the existing file test2.txt
>>>os.remove("test2.txt")
```