Lab 2 - Python Programming Basics

www.huawei.com

HUAWEI

Contents

if Statements

- Python supports three control structures: if, for, and while, but it does not support switch statements in the C language.
- In Python programming, if statements are used to control execution of control programs, and the basic form is:

```
if Judging condition 1:
        Statement 1…
elif Judging condition 2:
        Statement 2…
elif Judging condition 3:
        Statement 3…
else:
        Statement 4…
```

while Statements

- The while statement in the Python language is used to execute a loop program that, under certain conditions, loops through a program to handle the same tasks that need to be repeated.
- When the condition of a while statement is never a Boolean false, the loop will never end, forming an infinite loop, also known as a dead loop. You can use the break statement in a loop to force a dead loop to end.
- How to use a while statement:

```
count = 0

while (count < 9):
    print("The count is:", count)
    count = count + 1

print("Good bye!")
```

for Statements

- In the Python language, the for loop can traverse any items of a sequence, such as a list, a dictionary, or a string.
- The for statement is different from a traditional for statement. The former accepts an iterative object (such as a sequence or iterator) as its argument, and one element is iterated each time.

```
for num in nums:
    if num == 1:
        print(num+”---”)
    else if num == 2:
        print(num+”///”)
    else:
        print('break not triggered')
```

for Statements Con…

- Example:

```
>>> for i in range(0,10):
>>>print(i)
```

- Output:

```
0
1
2
3
4
5
6
7
8
9
```

for Statements Con…

- Example:

```
>>> a=[1,3,5,7,9]
>>> for i in a:
>>>print(i)
```

- Output:

```
1
3
5
7
9
```

Loop Nesting

- Python allows one loop to be nested in another loop.
- Syntax of for loop nesting:

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

- Syntax of while loop nesting:

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

break and continue

- A break statement ends the entire loop, and if a break statement is triggered, the loop else is not executed.
- A continue statement ends the ongoing iteration of the loop, and begins the next iteration.
- If you use a nested loop, the break statement stops executing the deepest loop and starts executing the next line of code.
- The continue statement tells Python to skip the remaining statements of the current loop to proceed to the next round of loops.
- Both the break and continue statements are available in the while and for loops.

Contents

HUAWEI

Python Functions

- A function is a code segment that is organized, reusable, and used to implement a single function or associated functions.
- Functions can improve the modularity of applications and reuse of code.
- Python provides a number of built-in functions, such as print(). You can also create your own functions, which are called user-defined functions.

Common built-in functions

```
>>> 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
```

Common built-in functions con…

- 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
```

Defining a Function

- Define a function with the following rules:
  - The function code block begins with a def keyword, followed by the function name and parentheses ().
  - Any incoming arguments and independent variables must be placed in the middle of the parentheses. Parentheses can be used to define arguments.
  - The first line of the function can selectively use the document string to hold the description of the function.
  - The function content starts with a colon and indents.
  - *return[expression]* ends a function, and selectively returns a value to the caller. Returning without an expression is equivalent to returning none.

Defining a Function Con…

- Example:

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

Calling a Function

- Defining a function only gives the function a name, specifies the arguments contained in the function, and the code block structure.
- After the basic structure of this function is complete, you can execute it through another function call, or you can execute it directly from the Python prompt.

```
# Define a function
def test(str):
    print(str)
    return str

# Call a function
test("I want to call a user-defined function!")
test("call the same function again")
```

Transferring Arguments

- In Python, a type belongs to an object, and a variable is of no type.

```
a = [1,2,3]
a = "Huawei"
```

- In the above code, [1,2,3] is the list type, "Huawei" is a string type, and the variable a is of no type, which is only a reference (a pointer) to an object, and can be a list type object, or a string type object.

Argument Types

- The following are the formal argument types you can use when calling a Python function:
  - Essential argument: The essential arguments must pass in the function in the correct order, and the number of arguments for calling should be the same as defined.
  - Keyword argument: Keyword arguments and functions are called closely, and function calling uses keyword arguments to determine the values of incoming arguments.
  - Default argument: When a function is called, if the value of the default argument is not transferred, it is considered that a default value is used.
  - Indefinite length argument: You may need a function to handle more arguments than those originally stated. These arguments are called indefinite arguments and are not named when they are stated.

Argument Types con…

- Keyword argument:

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

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

```
>>> 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'}
```

Argument Types con…

- 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'}
```

Argument Types con…

- Name keyword arguments:
  - If you want to restrict names of keyword arguments, you can name keyword arguments

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

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

- An error will be returned during calling if no argument name is introduced. In this case, the keyword argument can be default.

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

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

Argument Types con…

- Argument combination: Arguments must be defined in the order of required arguments, default arguments, changeable arguments, named keyword arguments, and keyword 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)
>>> 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}
```

Argument Types con…

- Example: con…

```
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)
```

```
>>> 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': '#'}
```

Anonymous Functions

- Python uses lambda to create anonymous functions:
  - lambda is only an expression, and its function body is much simpler than def.
  - The body of lambda is an expression, not a block of code. Only limited logic can be encapsulated in lambda expressions
  - A lambda function has its own namespace and cannot access arguments outside of its own argument list or in the global namespace.
  - Although a lambda function may seem to write only one line, it is not the same as the C or C ++ inline function, which is designed to call small functions without consuming stack memory and therefore increases operational efficiency.

Anonymous Functions

- Syntax of lambda function:

```
lambda arguments: expression
```

- Example:

```
# Program to show the use of lambda functions
double = lambda x: x * 2

print(double(5))
```

- Output:

```
10
```

Global Variables and Local Variables

- A variable defined within a function has a local scope and is called a local variable, and a variable defined beyond a function has a global scope and is called a global variable.
- Local variables can only be accessed within the stated function, and global variables are accessible within the entire program.
- When a function is called, all variable names stated within the function are added to the scope.

OS-Related Calling and Functions - sys

- System-related Information module sys.
- SYS.ARGV passes arguments from outside the program to the program.
- Sys.stdout, Sys.stdin, and Sys.stderr represent the standard input and output, the file object of error output, respectively.
- Sys.stdin.readline () reads a line from the standard input and sys.stdout.write ("a") outputs a on the screen.
- Sys.exit () exits the program.
- Sys.modules is a dictionary that holds all imported modules.
- Sys.platform Gets the running OS environment.
- Sys.path is a list that indicates the path to identify modules.

OS-Related Calling and Functions - os

- os.environ contains the mapping between environment variables; os.environ["Home"] can get the value of the environment variable Home.
- os.chdir (dir) changes the current working directory; os.chdir('d:\\outlook').
- os.getcwd() gets the current directory.
- os.getegid () gets a valid group ID; Id;os.getgid () gets a valid group ID.
- os.getuid () gets a user ID; os.geteuid() gets a valid user ID.
- os.setegid () os.setegid () Os.seteuid () Os.setuid ().
- os.getgruops () gets a list of user group names.
- os.getlogin () gets the user login name.
- os.getenv () gets the environment variable.
- os.putenv () sets the environment variable.
- os.umask () sets the current permission mask and returns the previous permission mask. It is valid in Unix and windows.
- os.system (cmd) uses system calls to run the cmd command.

Contents

HUAWEI

Object-Oriented Programming

- Object-oriented programming (OOP) is a program design philosophy. OOP takes objects as the basic units of a program, and an object contains data and functions that manipulate data.
- Process-oriented programming treats a computer program as a series of command sets, that is, the sequential execution of a set of functions. In order to simplify program design, process-oriented programming divides functions into sub functions, that is, to reduce the system complexity by cutting block functions into smaller functions.
- OOP treats computer programs as a collection of objects, and each object can receive messages from other objects and process them. The execution of a computer program is a series of messages passing between objects.
- In Python, all data types can be treated as objects, and objects can be customized. The custom object data type is the concept of class in OOP.

Object-Oriented Design Philosophy

- The idea of object-oriented design (OOD) derives from nature, because the concepts of class and instance are natural.
- Class is an abstract concept. For example, our definition of class “student” refers to the concept of student, and instances refer to specific students such as Bart Simpson and Lisa Simpson. Therefore, the object-oriented design philosophy is to abstract classes and create instances according to classes.
- OOD has a higher abstraction than function because a class contains data and methods of manipulating data.

Relationship Between OOD and OOP

- OOD does not specifically require OOP languages. Actually, OOD can be implemented by a purely structured language, such as C, but if you want to construct data types that have the nature and characteristics of objects, you need to do more work on the program. When OO features are built into a language, OOP will be more efficient.
- On the other hand, an OOP language does not necessarily force you to write OO-related programs. For example, C++ can be taken as "better C"; Java, in turn, requires that everything be classes, and that a source file corresponds to a class definition. In Python, however, classes and OOP are not necessary for daily programming. Although it was designed from the outset to be object-oriented and structured to support OOP, Python does not qualify or require you to write OO code in your application.
- OOP is a powerful tool, and no matter whether you are ready to enter, learn, transition, or turn, OOP can be arbitrarily controlled.

Common Python OOP Terms

- Abstract/Implementation
- Encapsulation/Interface
- Composition
- Derivation/Inheritance/Inheritance Structure
- Generalization/Specialization
- Polymorphism
- Introspection/Reflection

Classes

- A class is a data structure that can be used to define objects, which combine data values with behavioral characteristics. A class is a real-world abstract entity that appears in a programmatic fashion. Instances materialize these objects. As an analogy, a class is a blueprint or a model used to produce real objects (instances).
- In Python, a class statement is similar to a function statement, with a corresponding keyword in the first line, followed by a body of code as its definition, as follows:

```
def functionName(args):
    ‘function documentation string'
    function_suite

class ClassName(object):
    ‘Click class documentation string’
    class_suite
```

Classes con…

```
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!")

dog = Dog('Max',3)
dog.sit()
```

```
Max is now sitting
```

Classes con…

- **__init__()** function is called automatically every time the class is being used to create a new object.
- Use the **__init__()** function to assign values to object properties, or other operations that are necessary to do when the object is being created
- The **self** parameter is a reference to the current instance of the class, and is used to access variables that belongs to the class.

Inheritance

- Inheritance is a way to create a class. In Python, a class can inherit from one or more parent classes. The original class is called a base class or a superclass.
- If there are several classes, and the classes have common variable attributes and function properties, then you can extract these variable properties and function properties as the base class properties. The special variable properties and function properties are defined in this class so that the base class's variable properties and function properties are accessible only if the base class is inherited. This increases the scalability of code.
- Abstraction is the extraction of similar parts.
- A base class is a class that abstracts the properties that are common to multiple classes.

Inheritance con…

```
Student name is Ahmed
Age is 10
School name is ABC
```

- Example:

```
151 class person:
152     def __init__(self, **kw):
153         self.name = kw["name"]
154         self.age = kw["age"]
155     def printPerson(self):
156         print('Student name is',self.name)
157         print('Age is',self.age)
158 
159 class student(person):
160     def __init__(self,schoolname,**kw):
161         person.__init__(self,**kw)
162         self.schoolname=schoolname
163     def printStudent(self):
164         self.printPerson()
165         print('School name is',self.schoolname)
166 
167 student1= student("ABC", name= 'Ahmed',age=10)
168 student1.printStudent()
```

Composition and Derivation

- Once a class is defined, the goal is to use it as a module and embed the objects into your code, mixed with other data types and logic execution streams. There are two ways to use classes in your code.
- The first way is the composition, which allows different classes to be mixed and added to other classes to add functionality and code reusability. You can create an instance of your own class in a larger class and implement some other properties and methods to enhance the original class object.
- The other way is derivation, which means that a subclass derives a new property on the basis of inheriting the parent class. A subclass may have a unique object that its parent class does not have, or a subclass defines an object with a name repeated in the parent class. A subclass is also called a derivation class.
- When there are significant differences between classes, composition behaves well; but when you design the same class with different functions, derivation is a more reasonable choice.

Subclasses

- One of the more powerful aspects of OOP is the ability to use a well-defined class, extend it, or modify it without affecting other snippets of code that use existing classes in the system. OOD allows a class feature inheritance by a descendant class or subclass. These subclasses inherit their core properties from the base class (or ancestor class, superclass). Also, these derivations may be extended to multiple generations.
- Related classes in a hierarchical derivation (or vertically adjacent to a class tree diagram) are of parent and subclass relationships. These classes deriving from the same parent class (or horizontally adjacent to the class tree diagram) are sibling relationships. The parent class and all high-level classes are considered ancestors.

Privatization

- By default, properties are "public" in Python and can be accessed by the module in which the class resides and by other modules that import the module in which the class resides. Many OO languages add some visibility to the data, providing only the function to access its values.
- Most OO languages provide access control characters to qualify the access of member functions.
- Double underline (__)
  - Python provides a preliminary form for the privacy of class elements (properties and methods). Properties that start with a double underline are "confusing" at run time, and therefore direct access is not allowed. Actually, it will precede the first name with an underscore and a class name.
- Single underline (_)
  - For simple module-level privatization, you only need to use a single underline character before the property name. This prevents the properties of the module from being loaded with the "from mymodule import *". This is strictly based on the scope, and therefore this is also true for functions.

Privatization

ERROR

```
line 169, in <module>
    print(student1.__age)

AttributeError: 'student' object has
no attribute '__age'
```

```
151 class person:
152     def __init__(self, **kw):
153         self._name = kw["name"]
154         self.__age = kw["age"]
155     def printPerson(self):
156         print('Student name is',self._name)
157         print('Age is',self.__age)
158
159 class student(person):
160     def __init__(self,schoolname,**kw):
161         person.__init__(self,**kw)
162         self.schoolname=schoolname
163     def printStudent(self):
164         self.printPerson()
165         print('School name is',self.schoolname)
166
167 student1= student("ABC", name= 'Ahmed',age=10)
168 student1.printStudent()
169 print(student1.__age)
```

_name is protected attribute
__age is a private attribute

Privatization con…

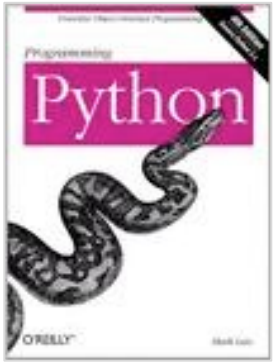
```
173 class JustCounter:
174     __secretCount = 0  # Private variable
175     publicCount = 0    # Public variable
176     def count(self):
177         self.__secretCount += 1
178         self.publicCount += 1
179         print(self.__secretCount)
180
181 counter = JustCounter()
182 counter.count()
183 counter.count()
184 print(counter.publicCount)
185 print(counter.__secretCount) # Error. Instance cannot access private variable
```

Hands-on

- Create Rectangle class which contain:
  - Width and length as a private attributes
  - Initialize the variables using name keyword arguments
  - Create an area method which return the area of the rectangle
- Create a rectangle
- Display its area

More Information

- Official site:
  - www.python.org
- References
  - Learning Python
  - Python Standard Library
  - Programming Python



Recommended for Learning

- Huawei e-learning site:
  - http://support.huawei.com/learning/Index!toTrainIndex
- Huawei knowledge base on the support website:
  - http://support.huawei.com/enterprise/servicecenter?lang=zh

Thanks

www.huawei.com