Hello World

In [1]:

```
print ('hello world') # Generate "hello world".
print("hello world") # Generate "hello world". The output is the same when single and doubl
```

```
hello world
hello world
```

Data Type

In [3]:

```
x=5.0 #built in data type
type(x)
```

Out[3]:

```
float
```

In [5]:

```
y='ahmed'
type(y)
```

Out[5]:

```
str
```

In [12]:

```
y=[1,2,"a"] #list
b=y[0:3] #last element excluded
c=tuple(y) # convert y to tuple
#print(c)
print(b)
len(y)
#type(c)
```

```
[1, 2, 'a']
```

Out[12]:

```
3
```

In [ ]:

```
True (reserved keyword)
```

number

In [18]:

```
print(True+False)# The output is 1. By default, True indicates 1, and False indicates 0.
print(True or False)# If True is displayed, enter or or perform the OR operation.
print(5//2)# The output is 2, and // is the rounding operator.
print(5%2)# The output is 1, and % is the modulo operator.
print(3**2) # The output is 9, and ** indicates the power operation.
print(5+1.26) # The output is 6.6. By default, the sum of numbers of different precisions i
```

```
1
True
2
1
9
6.26
```

Experimental Operations on lists

In [ ]:

```
list1=[1,'f',3.3,'a']
```

Check if the list is empty

In [21]:

```
#python
items=[]
if len(items) == 0:
    print("empty list")
else:
    print("It is not empty list")
```

```
empty list
```

In [4]:

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

```
empty list
```

Copy a list.

In [22]:

```
  File "<ipython-input-22-89d2ac70cc12>", line 1
    im t=3
       ^
SyntaxError: invalid syntax
```

In [23]:

```
#Method one:
old_list=["hello","Jeary"]
y=old_list[:]
#print(old_list[:])
print(y)
```

```
['hello', 'Jeary']
```

In [6]:

```
new_list = old_list[:]
print(new_list)
print(old_list)
```

```
['hello', 'Jeary']
['hello', 'Jeary']
```

In [24]:

```
old_list[0]
```

Out[24]:

```
'hello'
```

In [7]:

```
#Method two:
new_list = list(old_list)
print(new_list)
```

```
['hello', 'Jeary']
```

In [7]:

```
#Method three:
import copy
```

In [8]:

```
new_list1 = copy.copy(old_list)# copy
print(new_list1)
```

```
['hello', 'Jeary']
```

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.

In [26]:

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

Out[26]:

```
10
```

In [28]:

```
a[-1]
```

Out[28]:

```
10
```

In [45]:

```
b='''iam iman mostafa
working at .....
ghibjnmlk'''
print(b)
```

```
  File "<ipython-input-45-6830d91e0ad6>", line 1
    b='iam iman mostafa
                       ^
SyntaxError: EOL while scanning string literal
```

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.

In [208]:

```
items = [{'name': 'Homer', 'age': 39},{'name': 'Bart', 'age': 10},{"name": 'cater', 'age':
items.sort(key=lambda item: item.get("age"),reverse=True) #list.sort(key,reverse)
# lambda is anonymous function (function defined without a name)
#used when you require a nameless function
#have only one expression but can have any no. of arguments
print(items)
```

```
[{'name': 'Homer', 'age': 39}, {'name': 'cater', 'age': 20}, {'name': 'Bar
t', 'age': 10}]
```

In [8]:

```
help(items.sort)
```

```
Help on built-in function sort:

sort(...) method of builtins.list instance
    L.sort(key=None, reverse=False) -> None -- stable sort *IN PLACE*
```

Remove elements from a list.

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

In [35]:

```
a = [1,0, 2, 2, 3]
a.insert(7,9)
#print(a)
#a.remove(3)  #list.remove(obj)
#a.pop(0) #list.pop(index)
a
```

Out[35]:

```
[1, 0, 2, 2, 3, 9]
```

In [31]:

```
# ValueError will be returned if the removed element is not within the list.
a.remove(7)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-31-51fee2001d59> in <module>
      1 # ValueError will be returned if the removed element is not within t
he list.
----> 2 a.remove(7)

ValueError: list.remove(x): x not in list
```

Connect two lists.

In [44]:

```
#first method
listone = [1, 2, 3]
listtwo = [4, 5, 6]
mergedlist = listone + listtwo[0:2]
#print(mergedlist)
#second method
#listone.extend(listtwo)
#print(listone)
#append
#listone.append([1,23,3])
listone.append(listtwo)
print(listone)
```

```
[1, 2, 3, [4, 5, 6]]
```

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.

Define a tuple for an element.

In [18]:

```
t=(1,5,'hello')
print(t)
```

```
(1, 5, 'hello')
```

In [19]:

```
type(t)
```

Out[19]:

```
tuple
```

In [214]:

```
t2=(1,) # tuple
```

In [215]:

```
type(t2)
```

Out[215]:

```
tuple
```

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.

Define a 'changeable tuple'.

In [217]:

```
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], 'langu
age']
```

In [218]:

```
print(tmp[2])
```

```
(1, 2, 3)
```

In [219]:

```
print(tmp[0][1])
```

```
er
```

In [220]:

```
print(tmp[0][1][0])
```

```
e
```

Dictionaries

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

□ *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.

Create a dictionary

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

In [225]:

```
a = {'one': [1,2,3,4,5],'two': 2, 'three': 3}
print(a)
#print(a['one'])
#print(a['one'][-1]) #access with key not with index
```

```
{'one': [1, 2, 3, 4, 5], 'two': 2, 'three': 3}
```

In [226]:

```
 #a=[1,2,3] #list
# c=tuple(a) #cast
# print(c)
#another way to write dict
b = dict(one=[1,3], two=2, three=3)  #cast
print(b)
```

```
{'one': [1, 3], 'two': 2, 'three': 3}
```

In [227]:

```
#dict through list of tuples
c = dict([('one', 1), ('two', 2), ('three', 3)])
print(c)
```

```
{'one': 1, 'two': 2, 'three': 3}
```

In [10]:

```
#dict(zip([key1,key2,key3],[value1,value2,value3]))
d = dict(zip(['one', 'two', 'three'], [[1,7], 2, 3]))
print(d)
```

```
{'one': [1, 7], 'two': 2, 'three': 3}
```

In [42]:

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

```
{'one': 1, 'two': 2, 'three': 3}
```

In [43]:

```
print(a==b==c==d==e)
```

```
False
```

dictcomp

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

In [230]:

```
data = [("John","CEO",7),("Nacy","hr",7),("LiLei","engineer",9)]
#for name,work,age in data:
    # print(name)
     #print(work)
employee = {name:work for name, work,age in data}
print(employee)
```

```
{'John': 'CEO', 'Nacy': 'hr', 'LiLei': 'engineer'}
```

In [229]:

```
data = [("John","CEO",7),("Nacy","hr",7),("LiLei","engineer",9)]
for name,work,age in data:
    print(name)
    print(age)
```

```
John
7
Nacy
7
LiLei
9
```

Dictionary lookup

Look up directly according to a key value.
If there is no matched key value in a dictionary, KeyError is returned.

In [37]:

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

```
CEO
```

In [231]:

```
print(employee["Joh"])
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-231-85261c9baaeb> in <module>
----> 1 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.

In [17]:

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

```
hr
```

In [232]:

```
print(employee.get("Nac","wrong"))
```

```
wrong
```

In [234]:

```
# Three value assignment operations on dictionaries.
x = {'food':'Spam','quantity':4,'color':'pink'}
X =dict(food='Spam',quantity=4, color='pink')
x = dict([("food", "Spam"),("quantity", "4"),("color","pink")])
```

In [235]:

```
# dict.copy(): Copy data.
d =x.copy()
d['color'] = 'red'
print(x)                          # {'food':'Spam','quantity':4,'color':'pink'}
print(d)                          # {'food':'Spam','quantity':4,'color':'red'}
```

```
{'food': 'Spam', 'quantity': '4', 'color': 'pink'}
{'food': 'Spam', 'quantity': '4', 'color': 'red'}
```

In [21]:

```
# Element access.
print (d['name']) # Obtain the error information.
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-21-d401274b3d23> in <module>
      1 # Element access.
----> 2 print (d['name']) # Obtain the error information.

KeyError: 'name'
```

In [236]:

```
print(d.get('name')) # Output: None
print(d.get('name','The key value does not exist.'))    # Output: The key value does not ex
print(d.keys()) # Output: dict_keys(['food', 'quantity', 'color'])
print(x.values())# Output: dict_values(['Spam', 4, 'red'])
print(d.items())
```

```
None
The key value does not exist.
dict_keys(['food', 'quantity', 'color'])
dict_values(['Spam', '4', 'pink'])
dict_items([('food', 'Spam'), ('quantity', '4'), ('color', 'red')])
```

In [18]:

```
# Output: dict_items([('food', 'Spam'), ('quantity', 4), ('color', 'red')])
d.clear()# Clear all data in the dictionary.
print(d)# Output: {}
```

```
{}
```

In [19]:

```
del(d)# Delete the dictionary.
print(d)# The program is abnormal, and a message is displayed, indicating that d is not def
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-19-b1fd135d6891> in <module>
      1 del(d)# Delete the dictionary.
----> 2 print(d)# The program is abnormal, and a message is displayed, indic
ating that d is not defined.

NameError: name 'd' is not defined
```

string

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

Strings of Python: A string is a sequence composed of zero or multiple characters, and it is one of the sixth 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.

Single quotes and double quotes

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

In [26]:

```
s = 'string'
print(s)
```

```
string
```

In [27]:

```
ss="python string"
print(ss)
```

```
python string
```

In [28]:

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

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

Long strings

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

In [29]:

```
print('''"this is a long string",
jknklnkjnkljnmkjmn
hvjhvbjhbjhbjhbhe said''')
```

```
"this is a long string",he said
```

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:

In [62]:

```
 #rawStr = r'D:\SVN_CODE\V900R17C00_TRP\omu\src' #for path
rawStr= r'E:\Courses\Huwaie training\HCIA-AI V1.0 Mock Exam'
print(rawStr)
```

```
E:\Courses\Huwaie training\HCIA-AI V1.0 Mock Exam
```

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.

In [241]:

```
print("%c" % 95) #put the data in its ascii
#print("x= "+str(x))
```

```
_
```

In [249]:

```
print("%20.5f" % 2.5) #6 is no of spaces the no occupied
```

```
             2.50000
```

In [247]:

```
print("%20x" % 10) # x means hex
```

```
                   a
```

In [252]:

```
print("%.*f" % (12, 1.5))
```

```
1.500000000000
```

Connect and repeat strings

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

In [255]:

```
s = 'I' + ' ' + 'Knew' +' '+ 'Python' + '.'
print(s)
```

```
I Knew Python.
```

In [257]:

```
ss='I love Python.\n'*5
print(ss)
```

```
I love Python.
I love Python.
I love Python.
I love Python.
I love Python.
```

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.

In [31]:

```
del ss
print(ss)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-31-98639b0ddb82> in <module>
      1 del ss
----> 2 print(ss)

NameError: name 'ss' is not defined
```

Conditional and Looping Statements

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

If Statement

In [261]:

```
#Determine the entered score.
# input(): Receive input data.
score = int(input("Please enter your score.")) # The input function receives input, which i
#score = float(score)# Convert the score to a number.
# try:… except Exception:… is a Python statement used to capture exceptions. If an error oc
#indentation
try:
    if 100>=score>=90:        # Check whether the entered value is greater than the score o
        print("Excellent") # Generate the level when conditions are met.
    elif 90 > score >= 80:    #not elseif
        print("Good")
    elif 80>score>50:
        print("Medium")
    else:
        print("Bad")

except Exception:
    print("Enter a correct score.")
```

```
Please enter your score.20
Bad
```

"for" looping

In [265]:

```
for i in range(1,9,2): #range(start,stop,step)
    print(i)
```

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

In [269]:

```
a=[1,3,5,7,9]
for i in a[0:3]: #iterate on apart of list
    print(i)
#print(a)
```

```
1
3
5
7
9
```

In [273]:

```
for i in range(1,10):# Define the outer loop.
    for j in range(1,i+1):# Define the inner loop.
    # Format the output character string to align the generated result. The end attribute i
        print("%d*%d=%d"%(i,j,i*j), end=" ")
    print()
```

```
1*1=1*
2*1=2* 2*2=4*
3*1=3* 3*2=6* 3*3=9*
4*1=4* 4*2=8* 4*3=12* 4*4=16*
5*1=5* 5*2=10* 5*3=15* 5*4=20* 5*5=25*
6*1=6* 6*2=12* 6*3=18* 6*4=24* 6*5=30* 6*6=36*
7*1=7* 7*2=14* 7*3=21* 7*4=28* 7*5=35* 7*6=42* 7*7=49*
8*1=8* 8*2=16* 8*3=24* 8*4=32* 8*5=40* 8*6=48* 8*7=56* 8*8=64*
9*1=9* 9*2=18* 9*3=27* 9*4=36* 9*5=45* 9*6=54* 9*7=63* 9*8=72* 9*9=81*
```

In [272]:

```
print('Hello world',end='  ')
print('iman')
```

```
Hello world  iman
```

"while" looping

In [27]:

```
i=90
while (i<100):
    i+=1
    print(i)
```

```
91
92
93
94
95
96
97
98
99
100
```

In [277]:

```
i = 0# Create variable i.
while i<9:                                          # Set a condition for the loop.
    i+=1 # The value of i increases by 1 in each loop.
    if i == 3:                                        # Check whether the conditions are met.
        #print("Exit this loop.")
        continue# Execute continue to exit the current loop.
    if i == 5:
        #print("Exit the current big loop.")
        break# Exit the current big loop.
    print(i)
```

```
1
2
4
```

Functions

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

Common built-in functions

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

In [52]:

```
int('123')
```

Out[52]:

```
123
```

In [53]:

```
int(12.34)
```

Out[53]:

```
12
```

In [54]:

```
float('12.34')
```

Out[54]:

```
12.34
```

In [55]:

```
str(1.23)
```

Out[55]:

```
'1.23'
```

In [56]:

```
str(100)
```

Out[56]:

```
'100'
```

In [57]:

```
bool(1)
```

Out[57]:

```
True
```

In [58]:

```
bool('')
```

Out[58]:

```
False
```

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 name.

In [59]:

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

Out[59]:

```
1
```

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.

In [95]:

```
def my_abs(x):
    if x>=0:
        return x
    else:
        return -x
my_abs(3)
```

Out[95]:

```
3
```

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 "is instance()".

In [104]:

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

In [105]:

```
my_abs(3.5)
```

Out[105]:

```
3.5
```

In [31]:

```
def my_abs(x):
```

```
  File "<ipython-input-31-b37075c82ddc>", line 1
    def my_abs(x):
                  ^
SyntaxError: unexpected EOF while parsing
```

In [32]:

```
def my_abs(x):
    pass
```

In [108]:

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

In [109]:

```
my_abs('3')
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-109-9facf81466ad> in <module>
----> 1 my_abs('3')

<ipython-input-108-00d78b021b91> in my_abs(x)
      1 def my_abs(x):
      2     pass
----> 3     if x >= 0:
      4         return x
      5     else:

TypeError: '>=' not supported between instances of 'str' and 'int'
```

In [123]:

```
z=10
def fun():
    global z
    print(z)

fun() #calling funct
```

```
10
```

In [124]:

```
x=3
def fun():
    global x
    x=x+2
    print(x)
fun()
```

```
5
```

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.

In [21]:

```
def person(name, age, **kw): #** for optional argument
    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.

In [23]:

```
person('Michael', 30)
person(age=22,name="yyy") #useful if u have alot of arguments
```

```
name: Michael age: 30 other: {}
name: yyy age: 22 other: {}
```

In [55]:

```
#You can also input any number of keyword arguments.
person('Bob', 35, city='Beijing')
```

```
name: Bob age: 35 other: {'city': 'Beijing'}
```

In [56]:

```
 person('Adam', 45, gender='M', job='Engineer')
```

```
name: Adam age: 45 other: {'gender': 'M', 'job': 'Engineer'}
```

You can assemble a dictionary and convert it into keyword arguments as inputs. This is similar to assembling changeable arguments.

In [57]:

```
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 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.

In [58]:

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

```
name: Jack age: 24 other: {'city': 'Beijing', 'job': 'Engineer'}
```

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:

In [126]:

```
def add(*args):
    x=0
    for i in args:
        x=x+i
    print(x)
add(3,4)
```

```
7
```

In [196]:

```
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:

In [197]:

```
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.

In [54]:

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

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

In [55]:

```
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 "*".

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:

In [129]:

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

In [130]:

```
#Python interpreter will input the matched arguments according to argument positions and na
f1(1, 2)
```

```
a = 1 b = 2 c = 0 args = () kw = {}
```

In [61]:

```
f1(1, 2, c=3)
```

```
a = 1 b = 2 c = 3 args = () kw = {}
```

In [64]:

```
f1(1, 2, 3, 4, 5,6)
```

```
a = 1 b = 2 c = 3 args = (4, 5, 6) kw = {}
```

In [74]:

```
f1(1, 2, 3, 'a', 'b', x=99)
```

```
a = 1 b = 2 c = 3 args = ('a', 'b') kw = {'x': 99}
```

In [131]:

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

In [132]:

```
f2(1, 2, 99,d=3, ext='a')
```

```
a = 1 b = 2 c = 99 d = 3 kw = {'ext': 'a'}
```

In [48]:

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

In [49]:

```
#The most amazing thing is that you can call the above-mentioned function through a tuple a
args = (1, 2, 3, 4,6)
kw = {'d': 99, 'x': '#'}
f1(*args, **kw)
```

```
a = 1 b = 2 c = 3 args = (4, 6) kw = {'d': 99, 'x': '#'}
```

In [50]:

```
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.

Recursive function

You need to prevent a stack overflow when you use a recursive function. Functions are called through the stack 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.

In [103]:

```
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)
```

In [101]:

```
fact(3)
```

Out[101]:

```
6
```

In [81]:

```
fact_iter(1,8)
```

Out[81]:

```
8
```

In [105]:

```
fact_iter(2,3)
```

Out[105]:

```
6
```

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.

Object-Oriented Programming

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

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:

In [28]:

```
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") #title return capitalized first characte
    def roll_over(self):
        """Simulate rolling over when a dog is ordered to do so"""
        print(self.name.title()+" rolled over!")
```

In [29]:

```
dog=Dog()
dog.init("luc",2)
```

In [30]:

```
dog.sit()
```

```
Luc is now sitting
```

Access attributes

Let us see a complete instance.

In [77]:

```
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"

In [78]:

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

"Create the second object of the employee class"

In [81]:

```
emp2 = Employee("Manni", 5000)
emp3 = Employee("Alibaba", 330)
emp1.displayEmployee()
emp2.displayEmployee()
emp3.displayEmployee()
displayCount()
#print("Total Employee %d" % Employee.empCount)
```

```
Name :  Zara , Salary:  2000
Name :  Manni , Salary:  5000
Name :  Alibaba , Salary:  330

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-81-9b220a4ecb8b> in <module>
      4 emp2.displayEmployee()
      5 emp3.displayEmployee()
----> 6 displayCount()
      7 #print("Total Employee %d" % Employee.empCount)

NameError: name 'displayCount' is not defined
```

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.

In [87]:

```
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)
```

In [88]:

```
class Child(Parent): # Define a sub-class
    def __init__(self):
        print("Call sub-class construction method")
    def childMethod(self):
        print('Call sub-class method')
```

In [89]:

```
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
```

```
Call sub-class construction method
Call sub-class method
Call parent class method
Parent class attribute 200
```

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.

In [90]:

```
class JustCounter:
    __secretCount = 0  # Private variable
    publicCount = 0    # Public variable
    def count(self):
        self.__secretCount += 1
        self.publicCount += 1
        print(self.__secretCount)
```

In [91]:

```
counter = JustCounter()
counter.count()
counter.count()
print(counter.publicCount)
print(counter.__secretCount)  # Error. Instance cannot access private variable
```

```
1
2
2

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-91-097581c5fb47> in <module>()
      3 counter.count()
      4 print(counter.publicCount)
----> 5 print(counter.__secretCount)  # Error. Instance cannot access privat
e variable

AttributeError: 'JustCounter' object has no attribute '__secretCount'
```

Date and Time

This experiment mainly introduces related knowledge units about date and time in Python, and related operations on them.
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.
mThe time module of Python has many functions to convert common date formats.

In [92]:

```
## Get the current time
```

In [5]:

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

```
Local time: time.struct_time(tm_year=2020, tm_mon=4, tm_mday=22, tm_hour=7,
tm_min=12, tm_sec=49, tm_wday=2, tm_yday=113, tm_isdst=0)
```

Get the formatted time

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

In [110]:

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

```
Local time : Tue Aug 27 12:32:21 2019
```

In [95]:

```
## Format date
```

In [111]:

```
import time
# Format into 2016-03-20 11:45:39
print(time.strftime("%Y-%m-%d %H:%M:%S", time.localtime()))
```

```
2019-08-27 12:32:42
```

In [97]:

```
# Format into Sat Mar 28 22:24:24 2016
print(time.strftime("%a %b %d %H:%M:%S %Y", time.localtime()))
```

```
Thu Aug 15 22:24:17 2019
```

In [98]:

```
# 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")))
```

```
1459175064.0
```

Get calendar of a month

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

In [83]:

```
import calendar
cal = calendar.month(2020,2)
print("output calendar of January 2019:")
print(cal)
```

```
output calendar of January 2019:
   February 2020
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
```

In [114]:

```
calendar.prcal(2019)
```

```
                                  2019

      January               February                March
Mo Tu We Th Fr Sa Su  Mo Tu We Th Fr Sa Su  Mo Tu We Th Fr Sa Su
    1  2  3  4  5  6                  1  2  3                  1  2  3
 7  8  9 10 11 12 13   4  5  6  7  8  9 10   4  5  6  7  8  9 10
14 15 16 17 18 19 20  11 12 13 14 15 16 17  11 12 13 14 15 16 17
21 22 23 24 25 26 27  18 19 20 21 22 23 24  18 19 20 21 22 23 24
28 29 30 31           25 26 27 28           25 26 27 28 29 30 31

       April                 May                   June
Mo Tu We Th Fr Sa Su  Mo Tu We Th Fr Sa Su  Mo Tu We Th Fr Sa Su
 1  2  3  4  5  6  7            1  2  3  4  5                  1  2
 8  9 10 11 12 13 14   6  7  8  9 10 11 12   3  4  5  6  7  8  9
15 16 17 18 19 20 21  13 14 15 16 17 18 19  10 11 12 13 14 15 16
22 23 24 25 26 27 28  20 21 22 23 24 25 26  17 18 19 20 21 22 23
29 30                 27 28 29 30 31        24 25 26 27 28 29 30

        July                 August              September
Mo Tu We Th Fr Sa Su  Mo Tu We Th Fr Sa Su  Mo Tu We Th Fr Sa Su
 1  2  3  4  5  6  7               1  2  3  4                     1
 8  9 10 11 12 13 14   5  6  7  8  9 10 11   2  3  4  5  6  7  8
15 16 17 18 19 20 21  12 13 14 15 16 17 18   9 10 11 12 13 14 15
22 23 24 25 26 27 28  19 20 21 22 23 24 25  16 17 18 19 20 21 22
29 30 31              26 27 28 29 30 31     23 24 25 26 27 28 29
                                            30

      October              November              December
Mo Tu We Th Fr Sa Su  Mo Tu We Th Fr Sa Su  Mo Tu We Th Fr Sa Su
    1  2  3  4  5  6                  1  2  3                     1
 7  8  9 10 11 12 13   4  5  6  7  8  9 10   2  3  4  5  6  7  8
14 15 16 17 18 19 20  11 12 13 14 15 16 17   9 10 11 12 13 14 15
21 22 23 24 25 26 27  18 19 20 21 22 23 24  16 17 18 19 20 21 22
28 29 30 31           25 26 27 28 29 30     23 24 25 26 27 28 29
                                            30 31
```

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:

In [135]:

```
import re
print(re.match('www', 'www.runoob.com').span())  # Match at start
print(re.search('com', 'www.runoob.com'))           # Match not at start #search for com im
```

```
(0, 3)
<re.Match object; span=(11, 14), match='com'>
```

Differences between re.match and re.search

re.match 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.

In [167]:

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

```
No match!!
```

In [171]:

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

```
No match!!
```

File Manipulation

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

Read keyboard input

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

input( ) function:

The input([prompt]) can receive a Python expression as the input and return the result.

In [121]:

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

```
Please input:
Your input is:
```

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.

In [87]:

```
# Open a file
fo = open("test.txt", "w")
print("File name: ", fo.name)
print("closed or not: ", fo.closed)
print("access mode:", fo.mode)
```

```
File name:  test.txt
closed or not:  False
access mode: w
```

In [123]:

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

```
File name:  foo.txt
```

In [124]:

```
# Close the opened file
fo.close()
```

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.

In [178]:

```
# Open a file
fo = open(r'E:\Courses\Huwaie training\Huawei\HCIA.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
#www.baidu.com!
#Very good site!
```

Read a file

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

In [185]:

```
# Open a file
fo = open("E:\Courses\Huwaie training\Huawei\HCIA.txt", "r+")
str = fo.read(10) #read 10 characters from the file
print("The read string is: ", str)
```

```
The read string is:  www.baidu.
```

In [189]:

```
# Close an opened file
fo.close()
```

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)

In [195]:

```
mport os
 Rename file test1.txt to test2.txt
s.rename(r"E:\Courses\Huwaie training\Huawei\HCIA.txt", r"E:\Courses\Huwaie training\Huawei\
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
<ipython-input-195-8e0af3ded92f> in <module>
      1 import os
      2 # Rename file test1.txt to test2.txt
----> 3 os.rename(r"E:\Courses\Huwaie training\Huawei\HCIA.txt", r"E:\Course
s\Huwaie training\Huawei\test.txt" )

FileNotFoundError: [WinError 2] The system cannot find the file specified:
 'E:\\Courses\\Huwaie training\\Huawei\\HCIA.txt' -> 'E:\\Courses\\Huwaie tr
aining\\Huawei\\test.txt'
```

In [129]:

```
# Open a file
fo = open("test2.txt", "r+")
str = fo.read(10)
print("The read string is: ", str)
fo.close()
```

```
The read string is:  www.baidu.
```

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)

In [130]:

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

In [131]:

```
open("test2.txt")
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
<ipython-input-131-211f32a1b1d0> in <module>()
----> 1 open("test2.txt")

FileNotFoundError: [Errno 2] No such file or directory: 'test2.txt'
```