

# Defining a Function

- Begin with the keyword **def** followed by the function name and parentheses ( **( )** ). → `def abc():`
  - Any input parameters or arguments should be placed within these parentheses.
- The first statement of a function can be an optional statement
  - the documentation string of the *function* or *docstring*.
- The code block within every function starts with a colon ( : ) and is indented.
- The statement `return [expression]` exits a function, optionally passing back an expression to the caller.
  - A return statement with no arguments is the same as `return None`.

# Defining a Function

```
def functionname( parameters ):  
    "function_docstring"  
    function_suite  
    return [expression]
```

```
def printme( str ):  
    "This prints a passed string into this function"  
    print str  
    return
```

```
#!/usr/bin/python  
  
# Function definition is here  
def printme( str ):  
    "This prints a passed string into this function"  
    print str;  
    return;
```

```
# Now you can call printme function  
printme("I'm first call to user defined function!");  
printme("Again second call to the same function");
```

I'm first call to user defined function!  
Again second call to the same function



# Functions

- `def print_hello():# returns nothing`  
`print (hello)`
- `def gcd(m, n):`  
`if n == 0:`  
 `return m # returns m`  
`else:`  
 `return gcd(n, m % n) recursive call`
- `def has_args(arg1,arg2=['e', 0]):`      `# returns [9.16,[1,'b','a',7]]`  
 `num = arg1 + 4`  
 `mylist = arg2 + ['a',7]`  
 `return [num, mylist]`  
`has_args(5.16,[1,'b'])`

```
[9.16, [1, 'b', 'a', 7]]
```

# Functions

```
def test(x, y, z):  
    return x+1, y+1, z+1  
  
a, b, c = test(1, 2, 3)    # 賦值給「同樣數量」的變數  
print(a)    # 2  
print(b)    # 3  
print(c)    # 4
```

2  
3  
4

```
def test(x, y, z):  
    return x+1, y+1, z+1  
  
a = test(1, 2, 3)  
print(a)
```

(2, 3, 4)

tuple

```
def hello(n, msg):  
    def h1():        # 內部函式  
        return msg  
    def h2():        # 內部函式  
        return msg*2  
    if n == 1:  
        print(h1())  
    if n == 2:  
        print(h2())  
hello(1, 'ok')    # ok  
hello(2, 'ok')    # okok  
print(h2())        # 發生錯誤 name 'h2' is not defined
```

ok  
okok

Traceback (most recent call last):

File "C:\Users\user\.spyder-py3\temp.py", line 94, in <module>  
 print(h2()) # 發生錯誤 name 'h2' is not defined

NameError: name 'h2' is not defined

# Function Parameter

```
def changeme(mylist):  
    mylist.append([1,2,3,4]);  
    print("Values inside the function: ", mylist)  
    return  
  
mylist = [10, 20, 30]  
changeme(mylist)  
print("Values outside the function: ", mylist)
```

```
Values inside the function: [10, 20, 30, [1, 2, 3, 4]]  
Values outside the function: [10, 20, 30, [1, 2, 3, 4]]
```

# Function Parameter

```
def changeme(mylist):  
    mylist = [1, 2, 3, 4];  
    print("Values inside the function: ", mylist)  
    return  
  
mylist = [10, 20, 30]  
changeme(mylist)  
print("Values outside the function: ", mylist)
```

```
Values inside the function: [1, 2, 3, 4]  
Values outside the function: [10, 20, 30]
```

# Example

- 如果變數的內容是串列、字典或集合，在處理「多個變數同時賦值」時容易會遇到陷阱，因為變數只是「標籤」，當多個變數同時指向一個串列、字典或集合時，**只要變數內容被修改 (並非使用等號給值)**，不論這個變數是全域還是區域變數，另外一個變數內容也會跟著更動。

```

a = []
b = a
a.append(1)
print(a)      # [1]
print(b)      # [1]    # 被影響

c = []
d = c
c = [1]
print(c)      # [1]
print(d)      # []    # 不受影響

```

```

[1]
[1]
[1]
[]

```

```

a = []
b = a
c = []
d = c

def f1():
    # global c          # 如果加上這行，f2 裡的 c 就會被影響
    a.append(1)
    c = [1]
    print(a) # [1]
    print(b) # [1]    # 被影響
    print(c) # [1]
    print(d) # []    # 不受影響

def f2():
    print(a) # [1]    # 被影響
    print(b) # [1]    # 被影響
    print(c) # []    # 不受影響，但如果 f1 加上 global c，此處就會被影響
    print(d) # []    # 不受影響

f1()
f2()

```

```

[1]
[1]
[1]
[]
[1]
[1]
[]
[]

```



# Global and Local Variable

```
a = 1 # 建立全域變數a

def fun():
    b = a # 建立區域變數b，將它設定為全域變數a的值
    a = 10 # 改變全域變數a的值
    b = c # 讀取全域變數c的值，將它設定給變數b
    print(b) # 顯示變數b的值

c = 3 # 建立全域變數c

fun() # 呼叫函式

print(a) # 顯示變數a的值
print(b) # 顯示變數b的值
print(c) # 顯示變數c的值
```

UnboundLocalError: local variable 'a' referenced before assignment

```
a = 1 # 建立全域變數a

def fun():
    global a, c # 宣告函式中會用到全域變數a和c
    a = 10 # 改變全域變數a的值
    c = 30 # 改變全域變數c的值

c = 3 # 建立全域變數c

fun() # 呼叫函式

print(a) # 顯示變數a的值
print(c) # 顯示變數c的值
```

10

30

# Global and Local Variable

```
a = 1    # 建立全域變數a

def fun():
    a = 10    # 建立區域變數a
    print(a)    # 這個a是區域變數還是全域變數？

fun()

print(a)    # 這個a是區域變數還是全域變數？
```

10  
1



```
a = []
b = a
c = []
d = c

def f1():
    global c          # 如果加上這行，f2 裡的 c 就會被影響
    a.append(1)
    c = [1]
    print(a)  # [1]
    print(b)  # [1]    # 被影響
    print(c)  # [1]
    print(d)  # []     # 不受影響

def f2():
    print(a)  # [1]    # 被影響
    print(b)  # [1]    # 被影響
    print(c)  # []     # 不受影響，但如果 f1 加上 global c，此處就會被影響
    print(d)  # []     # 不受影響

f1()
f2()
```

```
[1]
[1]
[1]
[]
[1]
[1]
[1]
[]
```

# Default Arguments

```
#!/usr/bin/python

# Function definition is here
def printinfo( name, age = 35 ):
    "This prints a passed info into this function"
    print "Name: ", name;
    print "Age ", age;
    return;

# Now you can call printinfo function
printinfo( age=50, name="miki" );
printinfo( name="miki" );
```

```
Name:  miki
Age   50
Name:  miki
Age   35
```

# Variable-length Arguments

- You may need to process a function for more arguments than you specified while defining the function.

```
def functionname([formal_args,] *var_args_tuple ):
    "function_docstring"
    function_suite
    return [expression]
```

```
#!/usr/bin/python

# Function definition is here
def printinfo( arg1, *vartuple ):
    "This prints a variable passed arguments"
    print "Output is: "
    print arg1
    for var in vartuple:
        print var
    return;

# Now you can call printinfo function
printinfo( 10 );
printinfo( 70, 60, 50 );
```

```
Output is:
10
Output is:
70
60
50
```

# Variable-length Arguments

symbol    Str.formatting

- \*args = **list** of arguments - as positional arguments
- \*\*kwargs = **dictionary** - whose keys become separate keyword arguments and the values become values of these arguments.

```
def print_everything(*args):  
    for count, thing in enumerate(args):  
        print('{0}. {1}'.format(count, thing))  
  
print_everything('apple', 'banana', 'cabbage')
```

```
0.apple  
1.banana  
2.cabbage
```

```
0,apple  
1,banana  
2,cabbage
```

```
def table_thing(**kwargs):  
    for name, value in kwargs.items():  
        print('{0}={1}'.format(name, value))  
  
table_thing(apple='fruit', cabbage='vegetable')
```

```
apple=fruit  
cabbage=vegetable
```

# \* Operator

```
a = [1,2,3,4,5]
b = (1,2,3,4,5)
c = {'x':1, 'y':2, 'z':3}
d = {'x', 'y', 'z'}
```

```
print(*a)      # 1 2 3 4 5
print(*b)      # 1 2 3 4 5
print(*c)      # x y z
print(*d)      # x y z
```

```
1 2 3 4 5
1 2 3 4 5
x y z
y z x
```

```
In [1]: a=[1,2,3,4,5]
```

```
In [2]: print(a)
[1, 2, 3, 4, 5]
```

```
In [3]: print(*a)
1 2 3 4 5
```

```
In [6]: c={1:'a',2:'b',3:'c',4:'d',5:'e'}
```

```
In [7]: print(c)
{1: 'a', 2: 'b', 3: 'c', 4: 'd', 5: 'e'}
```

```
In [8]: print(*c)
1 2 3 4 5
```

# string.format

%%	在字串 中顯示%
%d	以10 進位整數方式輸出
%f	將浮點 數以10進位方式輸出
%e, %E	將浮點 數以10進位方式輸出，並使用科學記號
%o	以8進 位整數方式輸出
%x, %X	將整 數以16進位方式輸出
%s	使用str() 將字串輸出
%c	以字元 方式輸出
%r	使用repr() 輸 出字串

```
In [6]: "example : %.2f" % 19.234
Out[6]: 'example : 19.23'
```

```
In [5]: "example : %8.2f" % 19.234
Out[5]: 'example : 19.23'
```



# string.format

<template>.format(<positional\_argument(s)>,<keyword\_argument(s)>)

- b.format(a) and format(a,b)

```
In [1]: '{}'.format('abc')
Out[1]: 'abc'
```

```
In [2]: '{{}}'.format('abc')
Out[2]: '{}'
```

```
In [3]: '{{{}}}'.format('abc')
Out[3]: '{abc}'
```

```
In [4]: '{{{{{}}}}'.format('abc')
Out[4]: '{{{}}}'
```

```
In [5]: '{{{{{{{}}}}}'.format('abc')
Out[5]: '{{{abc}}}'
```

```
1 >>> print('{} {}'.format('hello', 'world')) # 不帶欄位
2 hello world
```

```
3 >>> print('{0} {1}'.format('hello', 'world')) # 帶數字編號
4 hello world
```

```
5 >>> print('{0} {1} {0}'.format('hello', 'world')) # 打亂順序
6 hello world hello
```

```
7 >>> print('{1} {1} {0}'.format('hello', 'world'))
8 world world hello
```

```
9 >>> print('{a} {tom} {a}'.format(tom='hello', a='world')) # 帶關鍵字
10 world hello world
```

# string.format()

```
# ===== 向右對齊 =====  
print('{:>10}'.format('test'))
```

```
# ===== 向左對齊 =====  
print('{:10}'.format('test'))  
# 等同  
print('{:<10}'.format('test'))
```

```
# ===== 置中 =====  
print('{:^10}'.format('test'))
```

```
In [7]: print('{:>10}'.format('test'))  
      test  
  
In [8]: print('{:10}'.format('test'))  
test  
  
In [9]: print('{:^10}'.format('test'))  
      test
```

```
1 >>> print('{} and {}'.format('hello', 'world')) # 預設左對齊
```

```
2 hello and world
```

```
3 >>> print('{:10s} and {:>10s}'.format('hello', 'world')) # 取10位左對齊，取10位右對齊
```

```
4 hello and world
```

```
5 >>> print('{:^10s} and {:^10s}'.format('hello', 'world')) # 取10位中間對齊
```

```
6 hello and world
```

```
7 >>> print('{} is {:.2f}'.format(1.123, 1.123)) # 取2位小數
```

```
8 1.123 is 1.12
```

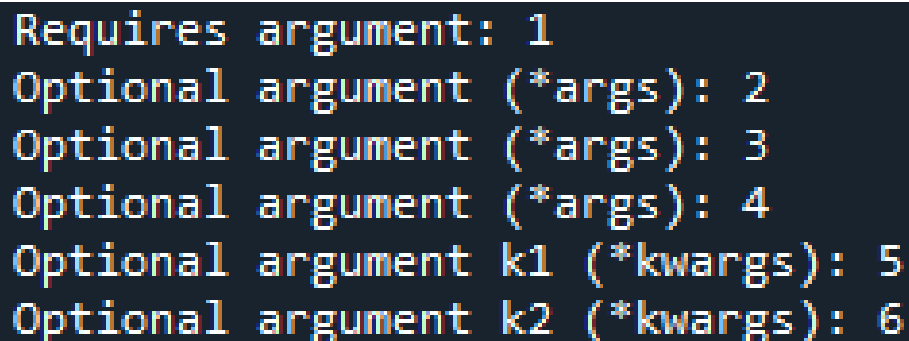
```
9 >>> print('{0} is {0:>10.2f}'.format(1.123)) # 取2位小數，右對齊，取10位
```

```
10 1.123 is 1.12
```

# Variable-length Arguments

- You can also use both in the same function definition but `*args` must occur before `**kwargs`.

```
def test_kwargs(first, *args, **kwargs):  
    print 'Required argument: ', first  
    for v in args:  
        print 'Optional argument (*args): ', v  
    for k, v in kwargs.items():  
        print 'Optional argument %s (*kwargs): %s' % (k, v)  
  
test_kwargs(1, 2, 3, 4, k1=5, k2=6)
```



```
Requires argument: 1  
Optional argument (*args): 2  
Optional argument (*args): 3  
Optional argument (*args): 4  
Optional argument k1 (*kwargs): 5  
Optional argument k2 (*kwargs): 6
```

# The *Anonymous* Functions

- You can use the *lambda* keyword to create **small** anonymous functions.
  - These functions are called **anonymous** because they are not declared in the standard manner by using the *def* keyword.
- The syntax of *lambda* functions contains only a single statement,

```
lambda [arg1 [,arg2,...argn]]:expression
```

```
#!/usr/bin/python

# Function definition is here
sum = lambda arg1, arg2: arg1 + arg2;

# Now you can call sum as a function
print "Value of total : ", sum( 10, 20 )
print "Value of total : ", sum( 20, 20 )
```

```
value of total : 30
value of total : 40
```

# Python Files I/O-Keyboard Input

- Python provides **one** built-in functions to read a line of text from standard input, which by default comes from the keyboard.
  - input
- The *input([prompt])* function → it assumes the input is a valid Python expression (include string) and returns the evaluated result to you.

# The *input* Function

```
#!/usr/bin/python  
  
str = input("Enter your input: ");  
print "Received input is : ", str
```

```
Enter your input:Hello  
Received input is : Hello
```

```
Enter your input:[x*5 for in range[2,10,2]]  
Received input is : [x*5 for in range[2,10,2]]
```

# Opening and Closing Files

- The file manipulation using a *file* object.
- *open* : Before you can read or write a file, you have to open it using Python's built-in *open()* function.
- This function creates a *file* object, which would be utilized to call other support methods associated with it.

# Open function

- Syntax `file object = open(file_name [, access_mode][, buffering])`
- **file\_name:** is a string value that contains the name of the file.
- **access\_mode:** determines the mode in which the file has to be opened, i.e., *read*, *write*, *append*, etc.
  - This is optional parameter and the default file access mode is read (r).
- **buffering:**
  - 0, no buffering takes place.
  - 1, line buffering is performed while accessing a file.
  - an integer greater than 1, then buffering action is performed with the indicated buffer size.
  - If negative, the buffer size is the system default (default behavior).



# access\_mode

Modes	Description		
r	Opens a file for reading only. The file pointer is placed at the beginning of the file. <u>This is the default mode.</u>	a	Opens a file for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
rb	<u>Opens a file for reading only in binary format.</u> The file pointer is placed at the beginning of the file. This is the default mode.	ab	Opens a file for appending in binary format. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
r+	<u>Opens a file for both reading and writing.</u> The file pointer is placed at the beginning of the file.	a+	Opens a file for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in append mode. If the file does not exist, it creates a new file for reading and writing.
rb+	<u>Opens a file for both reading and writing in binary format.</u> The file pointer is placed at the beginning of the file.	ab+	Opens a file for both appending and reading in binary format. The file pointer is at the end of the file if the file exists. The file opens in append mode. If the file does not exist, it creates a new file for reading and writing.
w	Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.		
wb	Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.		
w+	Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.		
wb+	Opens a file for both writing and reading in binary format. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.		

# The *file* Object Attributes

- Once a file is opened and you have one *file* object, you can get various information related to that file.

Attribute	Description
file.closed	Returns true if file is closed, false otherwise.
file.mode	Returns access mode with which file was opened.
file.name	Returns name of the file.
file.softspace	Returns false if space explicitly required with print, true otherwise.

Default = 0

Python 3 disable

# Example

```
f = open("f.txt", "wb")  
print("Name of the file:", f.name)  
print("Close or not:", f.closed)  
print("Opening mode:", f.mode)
```

```
Name of the file: f.txt  
Close or not: False  
Opening mode: wb
```

# The *close()* Function

- The `close()` method of a *file* object flushes any unwritten information and closes the file object, after which no more writing can be done.
- Python automatically closes a file when the reference object of a file is reassigned to another file.
  - To use the `close()` method to close a file.
- Syntax

```
fileObject.close();
```

```
f = open("f.txt", "wb")
print("Name of the file:", f.name)
print("Opening mode:", f.mode)
f.close()
print("Close or not:", f.closed)
```

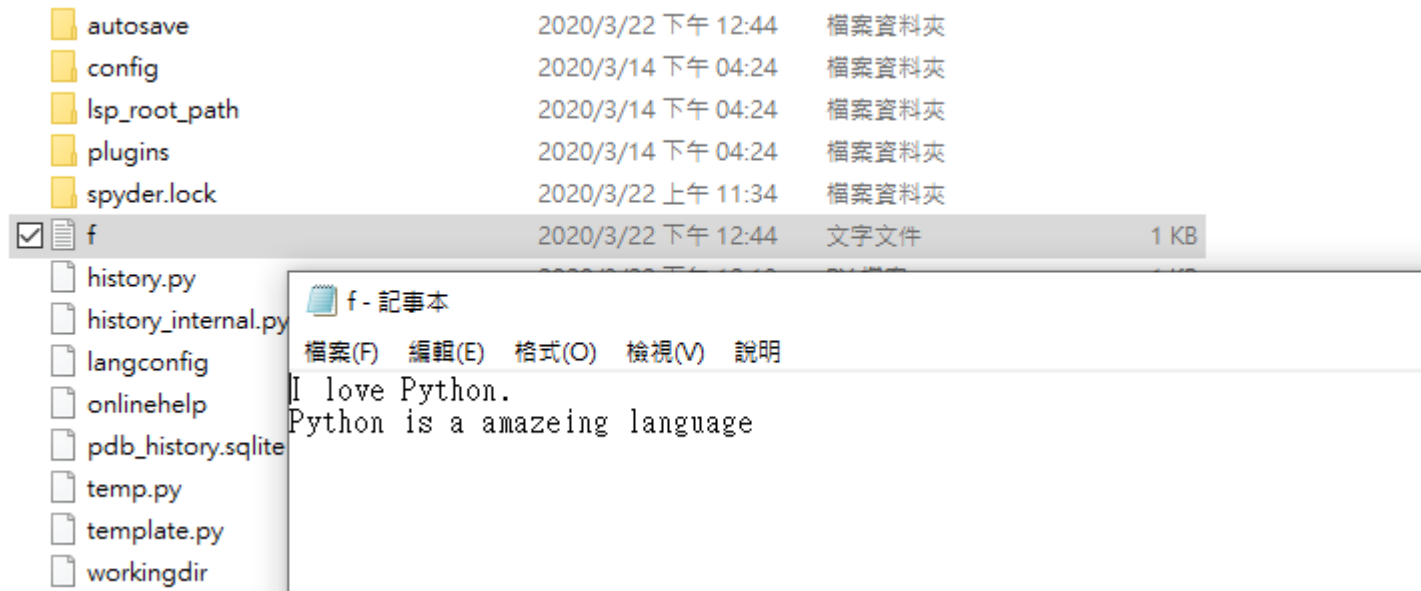
```
Name of the file: f.txt
Opening mode: wb
Close or not: True
```

# Reading and Writing Files

- The *file* object provides a set of access methods.
  - *read()* and *write()* methods to read and write files.

Syntax `fileObject.write(string);`

```
f = open("f.txt", "w")
f.write("I Love Python.\nPython is a amazeing Language\n")
f.close()
```



# The *read()* Method

- Syntax `fileObject.read([count]);`

```
f = open("f.txt", "r+")
str = f.read(10);
print("Read string is:", str)
f.close()
```

- Passed parameter is the number of bytes to be read from the opened file.

```
Read string is: I love Pyt
```

# File Positions

- The *tell()* method tells you the current position within the file.
    - The next read or write will occur at that many bytes from the beginning of the file.
  - The *seek(offset[, from])* method changes the current file position.
    - The *offset* indicates the number of bytes to be moved.
    - The *from* specifies the reference position from where the bytes are to be moved.
  - *From* is set to 0,
    - it means use the beginning of the file as the reference position
- 1: uses the current position as the reference position.
- 2: the end of the file would be taken as the reference position.

# Example

```
f = open("f.txt", "r+")
str = f.read(10);
print("Read string is:", str)
position = f.tell()
print("Current file position:",position)
position = f.seek(0,0)
str = f.read(10)
print("Again read string is:", str)

f.close()
```

```
Read string is: I love Pyt
Current file position: 10
Again read string is: I love Pyt
```



# Renaming and Deleting Files

- Python **os** module provides methods to perform file-processing operations, such as *renaming* and *deleting* files.
- The *rename()* Method














```
os.rename(current_file_name, new_file_name)
```














```
import os
os.rename("f.txt", "g.txt")
```

## The *remove()* Method

```
os.remove(file_name)
```

```
import os
os.remove("g.txt")
```

 autosave	2020/3/22 下午 03:23	檔案資料夾
 config	2020/3/14 下午 04:24	檔案資料夾
 lsp_root_path	2020/3/14 下午 04:24	檔案資料夾
 plugins	2020/3/14 下午 04:24	檔案資料夾
 spyder.lock	2020/3/22 上午 11:34	檔案資料夾
 g	2020/3/22 下午 12:44	文字文件
 history.py	2020/3/22 下午 12:13	PY 檔案
 history_internal.py	2020/3/22 上午 11:34	PY 檔案
 langconfig	2020/3/14 下午 04:24	檔案
 onlinehelp	2020/3/17 下午 09:59	檔案
 pdb_history.sqlite	2020/3/22 上午 11:34	SQLITE 檔案
 temp.py	2020/3/22 下午 03:23	PY 檔案
 template.py	2020/3/14 下午 04:24	PY 檔案

 autosave	2020/3/22 下午 03:23	檔案資料夾
 config	2020/3/14 下午 04:24	檔案資料夾
 lsp_root_path	2020/3/14 下午 04:24	檔案資料夾
 plugins	2020/3/14 下午 04:24	檔案資料夾
 spyder.lock	2020/3/22 上午 11:34	檔案資料夾
 history.py	2020/3/22 下午 12:13	PY 檔案
 history_internal.py	2020/3/22 上午 11:34	PY 檔案
 langconfig	2020/3/14 下午 04:24	檔案
 onlinehelp	2020/3/17 下午 09:59	檔案
 pdb_history.sqlite	2020/3/22 上午 11:34	SQLITE 檔案
 temp.py	2020/3/22 下午 03:25	PY 檔案
 template.py	2020/3/14 下午 04:24	PY 檔案
 workingdir	2020/3/22 下午 03:25	檔案















# Directories in Python

- The **os** module has several methods to create, remove, and change directories.

- The *mkdir()* Method

```
#!/usr/bin/python
import os














# Create a directory "test"
os.mkdir("test")
```

	autosave	2020/3/22 下午 03:23	檔案資料夾
	config	2020/3/14 下午 04:24	檔案資料夾
	lsp_root_path	2020/3/14 下午 04:24	檔案資料夾
	plugins	2020/3/14 下午 04:24	檔案資料夾
	spyder.lock	2020/3/22 上午 11:34	檔案資料夾
<input checked="" type="checkbox"/> 	test	2020/3/22 下午 03:27	檔案資料夾
	history.py	2020/3/22 下午 12:13	PY 檔案
	history_internal.py	2020/3/22 上午 11:34	PY 檔案
	langconfig	2020/3/14 下午 04:24	檔案
	onlinehelp	2020/3/17 下午 09:59	檔案
	pdb_history.sqlite	2020/3/22 上午 11:34	SQLITE 檔案
	temp.py	2020/3/22 下午 03:27	PY 檔案
	template.py	2020/3/14 下午 04:24	PY 檔案
	workingdir	2020/3/22 下午 03:27	檔案

- The *rmdir()* Method

```
#!/usr/bin/python
import os

# This would remove "/tmp/test" directory.
os.rmdir( "/tmp/test" )
```

	autosave	2020/3/22 下午 03:23	檔案資料夾
	config	2020/3/14 下午 04:24	檔案資料夾
	lsp_root_path	2020/3/14 下午 04:24	檔案資料夾
	plugins	2020/3/14 下午 04:24	檔案資料夾
	spyder.lock	2020/3/22 上午 11:34	檔案資料夾
	history.py	2020/3/22 下午 12:13	PY 檔案
	history_internal.py	2020/3/22 上午 11:34	PY 檔案
	langconfig	2020/3/14 下午 04:24	檔案
	onlinehelp	2020/3/17 下午 09:59	檔案
	pdb_history.sqlite	2020/3/22 上午 11:34	SQLITE 檔案
	temp.py	2020/3/22 下午 03:30	PY 檔案
	template.py	2020/3/14 下午 04:24	PY 檔案
	workingdir	2020/3/22 下午 03:30	檔案

# Directories in Python

- The *getcwd()* Method

```
#!/usr/bin/python
import os

# This would give location of the current directory
os.getcwd()
```

```
C:\Users\user\.spyder-py3
```

## The *chdir()* Method

```
#!/usr/bin/python
import os

# Changing a directory to "/home/newdir"
os.chdir("/home/newdir")
```

```
print(os.getcwd())
os.chdir("C:/Users/")
print(os.getcwd())
```

```
C:\Users\user\.spyder-py3
C:\Users
```



# Overview of OOP Terminology

- **Class:** A user-defined prototype for an object that defines a set of attributes to characterize the class.
  - The attributes are data members (class variables and instance variables) and methods, accessed via dot notation (.).
- **Data member:** includes class variables and instance variables that holds data associated with a class and its objects.
  - **Class variable:** A variable that is shared by all instances of a class.
    - Class variables are defined within a class but also outside any of the class's methods.
    - Static variable
    - Class variables aren't used as frequently as instance variables are.
  - **Instance variable:** A variable that is defined inside a method and belongs only to the current instance of a class.

# Creating Class

- The ***class*** statement creates a new class definition.

```
class ClassName:  
    'Optional class documentation string'  
    class_suite
```

- The class has a documentation string, which can be accessed via *ClassName.\_\_doc\_\_*.
- The *class\_suite* consists of all the component statements defining
  - data members 
    - Class variable
    - Instance variable
  - methods. 
    - Class method
    - Instance method
    - Static method

# EXAMPLE

- The variable `empCount` is a class variable whose value would be shared among all instances of this class.
  - This can be accessed as `Employee.empCount` from inside the class or outside the class.
- `__init__()` is a special method, which is called class **constructor** or **initialization method**.
  - Python **automatically calls** when you create a new instance of this class.
- You declare other class methods like normal functions with the exception that the first argument to each method is ***self***.
  - Python adds the ***self*** argument to the list for you; you don't need to include it when you call the methods.

```
class Employee:
    empCount = 0
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary
        Employee.empCount = Employee.empCount + 1

    def displayCount (self):
        print("Total Employee %d" % Employee.empCount)

    def displayEmployee (self):
        print("Name :", self.name, "Salary :", self.salary)
```

# Creating instance objects

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

```
"This would create first object of Employee class"  
emp1 = Employee("Zara", 2000)  
"This would create second object of Employee class"  
emp2 = Employee("Manni", 5000)
```

- Accessing attributes

```
emp1.displayEmployee()  
emp2.displayEmployee()  
print("Total Employee %d" % Employee.empCount)
```

# Example

```
class Employee:
    empCount = 0
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary
        Employee.empCount = Employee.empCount + 1

    def displayCount (self):
        print("Total Employee %d" % Employee.empCount)

    def displayEmployee (self):
        print("Name :", self.name, "Salary :", self.salary)

emp1 = Employee("Zara", 2000)
emp2 = Employee("Manni", 5000)

emp1.displayEmployee()
emp2.displayEmployee()
print("Total Employee %d" % Employee.empCount)
```

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



```
class human():
    def __init__(self):
        self.eye = 2
        self.ear = 2
        self.nose = 1
        self.mouth = 1
    def say(self, msg):
        print(msg)
    def play(self, thing):
        print(thing)

human.hand = 2    # 定義 hand 屬性
human.leg = 2     # 定義 leg 屬性

p = human()
print(p.hand)    # 2
print(p.leg)     # 2
```

2  
2

```
class human():
    def __init__(self):
        self.eye = 2
        self.ear = 2
        self.nose = 1
        self.mouth = 1
    def say(self, msg):
        print(f'{self.name} say: {msg}')    # 使用 self.name 取得 name 屬性的值
    def play(self, thing):
        print(thing)

p1 = human()
p1.name = 'Jack'    # 設定 name 屬性
p1.say('hello')    # Pa say: hello
```

Jack say: hello

```
class human():
    def __init__(self):
        self.eye = 2
        self.ear = 2
        self.nose = 1
        self.mouth = 1
    def say(self, msg):
        print(f'{self.name} say: {msg}')
    def play(self, thing):
        print(thing)
```

```
p1 = human()      # 定義 p1
p2 = human()      # 定義 p2
p1.name = 'Jack'  # p1 的名字叫做 Jack
p1.age = 18       # p2 的 age 為 18
```

```
p2.name = 'Rose'  # P2 的名字叫做 Rose
p2.weight = 50    # p2 的 weight 為 50
```

```
p1.say('hello')   # oxxo say: hello
print(p1.age)     # 18
p2.say('song')    # gkpen say: song
print(p2.weight)  # 50
```

```
Jack say: hello
18
Rose say: song
50
```

```
class human():
    def __init__(self):
        self.eye = 2
        self.ear = 2
        self.nose = 1
        self.mouth = 1
    def say(self, msg):
        print(f'{self.name} say: {msg}')
    def play(self, thing):
        print(thing)
```

```
p1 = human()
p1.play = '?????' # 覆寫 play 屬性
print(p1.play)    # ?????
```

```
?????
```

# @property

```
class a:
    def a(self):
        return 'aaaaa'
    @property
    def b(self):
        return 'bbbbbb'

oxxo = a()
oxxo.a = '12345'
print(oxxo.a)    # 12345
oxxo.b = '12345'
print(oxxo.b)    # 發生錯誤  can't set attribute
```

```
12345
Traceback (most recent call last):

  File "C:\Users\user\untitled0.py", line 314, in <module>
    oxxo.b = '12345'

AttributeError: can't set attribute
```

```
class Bank_acount:
    def __init__(self):
        self._password = '預設密碼 0000'
```

```
    @property
    def password(self):
        return self._password
```

```
    @password.setter
    def password(self, value):
        self._password = value
```

```
    @password.deleter
    def password(self):
        del self._password
        print('del complite')
```

```
andy = Bank_acount()
print(andy.password)
andy.password = '1234'
print(andy.password)
del andy.password
print(andy.password)
```

預設密碼 0000

1234

del complite

Traceback (most recent call last):

File "C:\Users\user\untitled0.py", line 342, in <module>  
 print(andy.password)

File "C:\Users\user\untitled0.py", line 325, in password  
 return self.\_password

AttributeError: 'Bank\_acount' object has no attribute '\_password'

# Built-In Class Attributes

- Every Python class keeps following [built-in attributes](#) and they can be accessed using **dot (.)** operator like any other attribute:
- **\_\_dict\_\_** : Dictionary containing the class's namespace.
- **\_\_doc\_\_** : Class documentation string or None if undefined.
- **\_\_name\_\_** : Class name.
- **\_\_module\_\_** : Module name in which the class is defined.
  - This attribute is "**\_\_main\_\_**" in interactive mode.
- **\_\_bases\_\_** : A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list.

# Example

```
class Employee:
    empCount = 0
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary
        Employee.empCount = Employee.empCount + 1

    def displayCount (self):
        print("Total Employee %d" % Employee.empCount)

    def displayEmployee (self):
        print("Name :", self.name, "Salary :", self.salary)

print("Employee.__doc__:", Employee.__doc__)
print("Employee.__name__:", Employee.__name__)
print("Employee.__module__:", Employee.__module__)
print("Employee.__bases__:", Employee.__bases__)
print("Employee.__dict__:", Employee.__dict__)
```

```
Employee.__doc__: None
Employee.__name__: Employee
Employee.__module__: __main__
Employee.__bases__: (<class 'object'>,)
Employee.__dict__: {'__module__': '__main__', 'empCount': 0, '__init__': <function Employee.__init__ at 0x0000018BF79041F8>, 'displayCount': <function Employee.displayCount at 0x0000018BF79045E8>, 'displayEmployee': <function Employee.displayEmployee at 0x0000018BF7904558>, '__dict__': <attribute '__dict__' of 'Employee' objects>, '__weakref__': <attribute '__weakref__' of 'Employee' objects>, '__doc__': None}
```

# Built-in Function *dir*

- The built-in function *dir* will give a list of names comprising the methods and attributes of an object.

```
print(dir(Employee))
```

```
['_class_', '_delattr_', '_dict_', '_dir_', '_doc_', '_eq_', '_format_', '_ge_', '_getattribute_', '_gt_', '_hash_', '_init_',  
'_init_subclass_', '_le_', '_lt_', '_module_', '_ne_', '_new_', '_reduce_', '_reduce_ex_', '_repr_', '_setattr_', '_sizeof_',  
'_str_', '_subclasshook_', '_weakref_', 'displayCount', 'displayEmployee', 'empCount']
```

- You can also get help using the help method: help (Exception).

# Destroying Objects (Garbage Collection)

- Python deletes unneeded objects (built-in types or class instances) automatically to free memory space.
- Python periodically reclaims blocks of memory that no longer are in use is termed garbage collection.
- Python's garbage collector runs during program execution and is triggered when an object's reference count reaches zero.
  - An object's reference count changes as the number of aliases that point to it changes.



# Destroying Objects

- An object's reference count increases when it's assigned a new name or placed in a container (list, tuple or dictionary).
- The object's reference count decreases when it's deleted with *del*, its reference is reassigned, or its reference goes out of scope.
- When an object's reference count reaches zero, Python collects it automatically.

```
a = 40          # Create object <40>
b = a          # Increase ref. count of <40>
c = [b]        # Increase ref. count of <40>

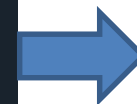
del a          # Decrease ref. count of <40>
b = 100        # Decrease ref. count of <40>
c[0] = -1      # Decrease ref. count of <40>
```

# EXAMPLE

- This `__del__()` destructor that prints the class name of an instance that is about to be destroyed.

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
    def __del__(self):
        class_name = self.__class__.__name__
        print(class_name, "destory")
```

```
pt1 = Point()
pt2 = pt1
pt3 = pt1
print(id(pt1), id(pt2), id(pt3))
del pt1
del pt2
del pt3
```



pt1 = None

```
1700665470152 1700665470152 1700665470152
Point destory
```

# Class Inheritance

- You can create a class by deriving it from a preexisting class by listing the parent class in parentheses after the new class name.

```
class SubClassName (ParentClass1[, ParentClass2, ...]):  
    'Optional class documentation string'  
    class_suite
```

- The child class inherits the attributes of the parent class
  - you can use those attributes as if they were defined in the parent class.
- A child class can also override data members and methods from the parent.

# EXAMPLE

```
#!/usr/bin/python

class Parent:          # define parent class
    parentAttr = 100
    def __init__(self):
        print "Calling parent constructor"

    def parentMethod(self):
        print 'Calling parent method'

    def setAttr(self, attr):
        Parent.parentAttr = attr

    def getAttr(self):
        print "Parent attribute :", Parent.parentAttr

class Child(Parent):    # define child class
    def __init__(self):
        print "Calling child constructor"

    def childMethod(self):
        print 'Calling child method'

c = Child()             # instance of child
c.childMethod()         # child calls its method
c.parentMethod()        # calls parent's method
c.setAttr(200)          # again call parent's method
c.getAttr()             # again call parent's method
```

```
Calling child constructor
Calling child method
Calling parent method
Parent attribute : 200
```

# Multiple Inheritance

```
class a():
    def says(self):
        print('a')

class b():
    def says(self):
        print('b')

class c(a, b):    # 先讀取 a 再讀取 b，就會將 a 裡的方法，覆寫 b 裡同名的方法
    pass

class d(b, a):    # 先讀取 b 再讀取 a，就會將 b 裡的方法，覆寫 a 裡同名的方法
    pass

c1 = c()
d1 = d()
c1.says()    # a
d1.says()    # b
```

a  
b

```
class grandpa():
    def __init__(self):
        self.eye = 2
        self.ear = 2
        self.nose = 1
        self.mouth = 1

class father(grandpa):
    def language(self):
        print('english')
    def skill(self):
        print('painting')

class son(father):
    def play(self):
        print('ball')

c1 = son()
print(c1.eye)    # 2
c1.skill()       # painting
c1.play()        # ball
```

2  
painting  
ball

```

class grandpa():
    def __init__(self):
        self.mouth = 1
    def __money(self):      # 建立一個私有方法 __money
        print('$1000')
    def getMoney(self):    # 建立一個 getMoney 的方法，執行私有方法 __money
        self.__money()

class father(grandpa):
    def skill(self):
        print('painting')

class son(father):
    def play(self):
        print('ball')

c1 = son()
c1.getMoney()           # $1000
c1.__money()            # 發生錯誤 'son' object has no attribute '__money'

```

```

$1000
Traceback (most recent call last):

  File "C:\Users\user\Inheritance.py", line 69, in <module>
    c1.__money()          # 發生錯誤 'son' object has no attribute '__money'
AttributeError: 'son' object has no attribute '__money'

```

# Super (Parent) Function (Python 3)

```
class Demo:
    __x = 0
    def __init__(self, i):
        self.__i = i
        Demo.__x += 1

    def __str__(self):
        return str(self.__i)

    def hello(self):
        print("hello " + self.__str__())

    def getx(cls):
        return cls.__x
```

```
class subDemo(Demo):
    def __init__(self, i, j):
        super().__init__(i)
        self.__j = j

    def __str__(self):
        return super().__str__() + "+" + str(self.__j)
```

```
a = subDemo(12, 34)
a.hello()
print("a.__x =", a.getx())
b = subDemo(56, 78)
b.hello()
print("b.__x =", b.getx())
print()
print("a.__x =", a.getx())
print("b.__x =", b.getx())
```

```
hello 12+34
a.__x = 1
hello 56+78
b.__x = 2

a.__x = 2
b.__x = 2
```

# Static Method & Class Method

```
class Demo:
    def __init__(self, i):
        self.i = i

    def __str__(self):
        return str(self.i)

    def hello(self):
        print("hello", self.i)
```

```
def statictest():
    print("this is static method..")
```

```
statictest = staticmethod(statictest)
```

```
def classtest(cls):
    print("this is class method..")
    print("the class name is", cls.__name__)
```

```
classtest = classmethod(classtest)
```

```
Demo.statictest()
Demo.classtest()
print()
a = Demo(9527)
a.hello()
a.statictest()
a.classtest()
```

```
this is static method..
this is class method..
the class name is Demo
```

```
hello 9527
this is static method..
this is class method..
the class name is Demo
```

```
@staticmethod
def statictest():
    print("this is static method..")
```

```
@classmethod
def classtest(cls):
    print("this is class method..")
    print("the class name is", cls.__name__)
```



# Static Method & Class Method

```
class Car:
    def __init__(self, vin):
        print('Ordinary __init__ was called for', vin)
        self.vin = vin
        self.brand = ''

    # 把 class method 使用在如果有 brand 資料時，作為替代的建構子
    # 其實仔細看一下，在 cls(vin) 時也會 call 到 __init__
    @classmethod
    def including_brand(cls, vin, brand):
        print('Class method was called')
        _car = cls(vin)
        _car.brand = brand
        return _car
```

```
car1 = Car('ABC123')
car2 = Car.including_brand('DEF123', 'Toyota')

print(car1.vin, car1.brand)
print(car2.vin, car2.brand)
```

```
Ordinary __init__ was called for ABC123
Class method was called
Ordinary __init__ was called for DEF123
ABC123
DEF123 Toyota
```

# Static Method & Class Method

比較項目	Class Methods	Static Methods
是否需要 cls 參數	Yes	No
是否有能力存取類別的狀態或是方法	Yes	No
標記方式	@classmethod	@staticmethod
用途	可做為建構子的替代與補充	僅做為使用的方法

# Instance, Class, Static Methods

```
# 汽車類別
class Cars:
    # 建構式
    def __init__(self):
        self.color = "blue"
    # 實體方法(Instance Method)
    def drive(self):
        print(self, 'is', self.color)
        self.message() # 呼叫其他方法
    # 實體方法(Instance Method)
    def message(self):
        print("Message method is called.")

mazda = Cars()

mazda.drive()
```

```
In [1]: runfile('C:/Users/user/untitled0.py', wdir='C:/Users/user')
<__main__.Cars object at 0x000001E1328CBB20> is blue.
Message method is called.
```

```
# 汽車類別
class Cars:
    # 建構式
    def __init__(self):
        self.color = "blue"
    # 實體方法(Instance Method)
    def drive(self):
        print(self, 'is', self.color)

Cars.drive()
```

```
In [2]: runfile('C:/Users/user/untitled0.py', wdir='C:/Users/user')
Traceback (most recent call last):

  File "C:\Users\user\untitled0.py", line 195, in <module>
    Cars.drive()

TypeError: drive() missing 1 required positional argument: 'self'
```

# Instance, Class, Static Methods

```
# 汽車類別
class Cars:
    door = 4 #類別屬性
    # 實體方法(Instance Method)
    def drive(self):
        self.__class__.door = 5
print("Cars original door: ", Cars.door)
mazda = Cars()
mazda.drive()
print("Cars new door: ", Cars.door)
```

```
In [3]: runfile('C:/Users/user/untitled0.py', wdir='C:/Users/user')
Cars original door: 4
Cars new door: 5
```

```
class Cars:
    door = 4 # 類別屬性
    # 類別方法(Class Method)
    @classmethod
    def open_door(cls):
        print(cls, "has %d doors" % cls.door)

mazda = Cars()

mazda.open_door() #透過物件呼叫

Cars.open_door() #透過類別呼叫
```

```
In [32]: runfile('C:/Users/user/untitled0.py', wdir='C:/Users/user')
<class '__main__.Cars'> has 4 doors
<class '__main__.Cars'> has 4 doors
```

**self.\_\_class\_\_** : 指向類別

# Instance, Class, Static Methods

```
# 汽車類別
class Cars:
    # 建構式
    def __init__(self, seat, color):
        self.seat = seat
        self.color = color
    # 廂型車
    @classmethod
    def van(cls):
        return cls(6, "black")
    # 跑車
    @classmethod
    def sports_car(cls):
        return cls(4, "yellow")
van = Cars.van()
print(van)
sports_car = Cars.sports_car()
print(sports_car)
```

```
# 汽車類別
class Cars:
    # 速率靜態方法
    @staticmethod
    def speed_rate(distance, minute):
        return distance / minute
# 透過物件呼叫
van = Cars()
van_rate = van.speed_rate(10000, 20)
print("van rate: ", van_rate)
# 透過類別呼叫
sports_car_rate = Cars.speed_rate(20000, 20)
print("sports car rate: ", sports_car_rate)
```

```
In [7]: runfile('C:/Users/user/untitled0.py', wdir='C:/Users/user')
van rate: 500.0
sports car rate: 1000.0
```

```
In [6]: runfile('C:/Users/user/untitled0.py', wdir='C:/Users/user')
<__main__.Cars object at 0x000001E132594EE0>
<__main__.Cars object at 0x000001E13293F280>
```

# Multiple Inheritance

```
class A:          # define your class A
.....

class B:          # define your class B
.....

class C(A, B):    # subclass of A and B
.....
```

- You can use **issubclass()** or **isinstance()** functions to check a relationships of two classes and instances.
- The **issubclass(sub, sup)** boolean function returns true if the given subclass **sub** is indeed a subclass of the superclass **sup**.
- The **isinstance(obj, Class)** boolean function returns true if obj is an instance of class Class is an instance of a subclass of

Class

```
In [9]: a=2

In [10]: isinstance(a,int)
Out[10]: True

In [11]: isinstance(a,str)
Out[11]: False

In [12]: isinstance(a,(str,int,list))
Out[12]: True
```

考到這 CH1 ~ 這  
Exception 考 (CH1)

# Inheritance

```
class CA():
    def __init__(self):
        self.var = 1

class CB(CA):
    def nothing(self):
        pass

class CC(CA):
    def __init__(self):
        CA.__init__(self)
        # do something else
        self.var2 = self.var + 1

class CD(CA):
    def __init__(self):
        pass

A = CA()
B = CB()
C = CC()
D = CD()

print("A.var:", A.var)
print("B.var:", B.var)
print("C.var:", C.var, "\tC.var2:", C.var2)
print("D.var:", D.var)
```

```
A.var: 1
B.var: 1
C.var: 1    C.var2: 2
Traceback (most recent call last):
```

```
File "C:\Users\user\繼承建構子教學.py", line 26, in <module>
    print("D.var:", D.var)
```

```
AttributeError: 'CD' object has no attribute 'var'
```

# Polymorphism

- *Polymorphism*, in the OOP, refers to the ability of an object to adapt the code to the type of the data.
- Polymorphism has *two* major applications in an OOP language.
  1. An object may provide different implementations of one of its methods depending on the type of the input parameters.
  2. Code written for a given type of data may be used on data with a derived type,
    - i.e., methods understand the class hierarchy of a type.



# Overriding Methods

- You can always override your parent class methods.

```
#!/usr/bin/python

class Parent:          # define parent class
    def myMethod(self):
        print 'Calling parent method'

class Child(Parent):   # define child class
    def myMethod(self):
        print 'Calling child method'

c = Child()            # instance of child
c.myMethod()           # child calls overridden method
```

Calling child method

# Example

- All animals "talk", but they have different "talk" behavior.
- The "talk" behavior is thus *polymorphic* in the sense that it is *realized differently depending on the animal*.
- The abstract "animal" concept does not actually "talk", but specific animals (like dogs and cats) have a concrete implementation of the action "talk".

```
class Animal:
    def __init__(self, name):
        self.name = name
    def talk(self):
        raise NotImplementedError("Subclass must implement abstract method")

class Cat(Animal):
    def talk(self):
        return "Meow"

class Dog(Animal):
    def talk(self):
        return "Woof Woof !!!"

animals = [Cat('Kitty'), Cat('Mr.Garfield'), Dog('Snoopy')]

for animal in animals:
    print(animal.name + ': ' + animal.talk())
```

```
Kitty: Meow
Mr.Garfield: Meow
Snoopy: Woof Woof !!!
```

# Base Overloading Methods

- Following table lists some generic functionality that you can override in your own classes.

SN	Method, Description & Sample Call
1	<code>__init__ ( self [,args...])</code> Constructor (with any optional arguments) Sample Call : <code>obj = className(args)</code>
2	<code>__del__( self )</code> Destructor, deletes an object Sample Call : <code>dell obj</code>
3	<code>__repr__( self )</code> Evaluatable string representation Sample Call : <code>repr(obj)</code>
4	<code>__str__( self )</code> Printable string representation Sample Call : <code>str(obj)</code>
5	<code>__cmp__ ( self, x )</code> Object comparison Sample Call : <code>cmp(obj, x)</code>

# Overloading Operators

- You could define the `__add__` method in your class to perform vector addition and then the plus operator would behave as per expectation

```
class Vector:
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def __str__(self):
        return 'Vector (%d, %d)' % (self.a, self.b)

    def __add__(self, other):
        return Vector(self.a + other.a, self.b + other.b)

v1 = Vector(2,10)
v2 = Vector(5,-2)

print(v1+v2)
```

Vector(7, 8)

**TypeError:** unsupported operand type(s) for +: 'Vector' and 'Vector'

# Overloading Operators

```
1 import math
2
3 class Circle:
4
5     def __init__(self, radius):
6         self.__radius = radius
7
8     def setRadius(self, radius):
9         self.__radius = radius
10
11    def getRadius(self):
12        return self.__radius
13
14    def area(self):
15        return math.pi * self.__radius ** 2
16
17    def __add__(self, another_circle):
18        return Circle( self.__radius + another_circle.__radius )
19
20 c1 = Circle(4)
21 print(c1.getRadius())
22
23 c2 = Circle(5)
24 print(c2.getRadius())
25
26 c3 = c1 + c2 # This became possible because we have overloaded + operator by addi
27 print(c3.getRadius())
```

1	4
2	5
3	9

OPERATOR	FUNCTION	METHOD DESCRIPTION
+	<code>__add__(self, other)</code>	Addition
*	<code>__mul__(self, other)</code>	Multiplication
-	<code>__sub__(self, other)</code>	Subtraction
%	<code>__mod__(self, other)</code>	Remainder
/	<code>__truediv__(self, other)</code>	Division
<	<code>__lt__(self, other)</code>	Less than
<=	<code>__le__(self, other)</code>	Less than or equal to
==	<code>__eq__(self, other)</code>	Equal to
!=	<code>__ne__(self, other)</code>	Not equal to
>	<code>__gt__(self, other)</code>	Greater than
>=	<code>__ge__(self, other)</code>	Greater than or equal to
[index]	<code>__getitem__(self, index)</code>	Index operator
in	<code>__contains__(self, value)</code>	Check membership
len	<code>__len__(self)</code>	The number of elements
str	<code>__str__(self)</code>	The string representation

```

1 import math
2
3 class Circle:
4
5     def __init__(self, radius):
6         self.__radius = radius
7
8     def setRadius(self, radius):
9         self.__radius = radius
10
11    def getRadius(self):
12        return self.__radius
13
14    def area(self):
15        return math.pi * self.__radius ** 2
16
17    def __add__(self, another_circle):
18        return Circle( self.__radius + another_circle.__radius )
19
20    def __gt__(self, another_circle):
21        return self.__radius > another_circle.__radius
22
23    def __lt__(self, another_circle):
24        return self.__radius < another_circle.__radius
25
26    def __str__(self):
27        return "Circle with radius " + str(self.__radius)
28
29 c1 = Circle(4)
30 print(c1.getRadius())
31
32 c2 = Circle(5)
33 print(c2.getRadius())
34
35 c3 = c1 + c2
36 print(c3.getRadius())
37
38 print( c3 > c2 ) # Became possible because we have added __gt__ method
39
40 print( c1 < c2 ) # Became possible because we have added __lt__ method
41
42 print(c3) # Became possible because we have added __str__ method

```

```

1 4
2 5
3 9
4 True
5 True
6 Circle with radius 9

```

# Data Hiding

- An object's attributes may or may not be *visible* outside the class definition.
- You can name attributes with a double underscore prefix, and those attributes will not be directly visible to outsiders.

```
#!/usr/bin/python

class JustCounter:
    __secretCount = 0

    def count(self):
        self.__secretCount += 1
        print self.__secretCount
```

```
counter = JustCounter()
counter.count()
counter.count()
print counter.__secretCount
```

1  
2

```
Traceback (most recent call last):
  File "test.py", line 12, in <module>
    print counter.__secretCount
AttributeError: JustCounter instance has no attribute '__secretCount'
```



# Data Hiding

- Python protects those members by internally changing the name to include the class name.
- You can access such attributes as *object.\_className\_attrName*.
- If you would replace your last line as following, then it would work for you:

```
.....  
print counter._JustCounter__secretCount
```

```
1  
2  
2
```