

Python Identifiers

1. Class names start with an uppercase letter and all other identifiers with a lowercase letter.
2. Starting an identifier with a single leading underscore (_) indicates by convention that the identifier is meant to be **private**.
3. Starting an identifier with two leading underscores (__) indicates a **strongly private identifier**.
4. If the identifier also ends with two trailing underscores, the identifier is a **language-defined special name**.

Multi-Line Statements

1. Python allows the use of the line continuation character (\) to denote that the line should continue

```
total = item_one + \
        item_two + \
        item_three
```

2. The semicolon (;) allows multiple statements on the single line given that neither statement starts a new code block

```
import sys; x = 'foo'; sys.stdout.write(x + '\n')
```

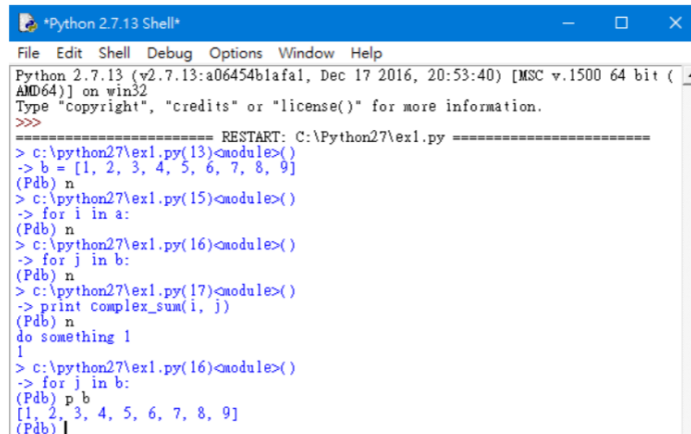
Python Debug (pdb)

```
#!/usr/bin/python
#本檔名為 pdb_example.py

import pdb #載入pdb模組
def complex_sum(x1, x2):
    print 'do something 1'
    value1 = 1 * x1
    value2 = 1 * x2
    return value1 + value2

a = [0, 1, 2, 3, 4, 5, 6, 7, 8]
pdb.set_trace() #中斷點
b = [1, 2, 3, 4, 5, 6, 7, 8, 9]

for i in a:
    for j in b:
        print complex_sum(i, j)
|
```



```
Python 2.7.13 Shell
File Edit Shell Debug Options Window Help
Python 2.7.13 (v2.7.13:a06454b1af1, Dec 17 2016, 20:53:40) [MSC v.1500 64 bit (AMD64)] on win32
Type "Copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:\Python27\ex1.py =====
> c:\python27\ex1.py(13)<module>()
-> b = [1, 2, 3, 4, 5, 6, 7, 8, 9]
(Pdb) n
> c:\python27\ex1.py(15)<module>()
-> for i in a:
(Pdb) n
> c:\python27\ex1.py(16)<module>()
-> for j in b:
(Pdb) n
> c:\python27\ex1.py(17)<module>()
-> print complex_sum(i, j)
(Pdb) n
do something 1
1
> c:\python27\ex1.py(16)<module>()
-> for j in b:
(Pdb) p b
[1, 2, 3, 4, 5, 6, 7, 8, 9]
(Pdb) |
```

q(quit): 離開

p [some variable](print): 秀某個變數的值

n(next line): 下一行

c(continue): 繼續下去

s(step into): 進入函式

r(return): 到本函式的return敘述式

l(list): 秀出目前所在行號

!: 改變變數的值

Standard Data Types

1. Numbers (Number data types store numeric values.)
2. String (Strings in Python are identified as a contiguous set of characters in between quotation marks(" ").)
3. List (Lists are the most versatile of Python's compound data types.)
4. Tuple (A tuple is another sequence data type that is similar to the list but it is immutable.)
5. Dictionary (Python's dictionaries are kind of hash table type.)
6. Set : frozenset([69, 6.9, 'str', True]) immutable –no duplicates & unordered

Python Strings

1. Subsets of strings can be taken using the slice operator (`[]` and `[:]`) with indexes **starting at 0** in the beginning of the string and working their way from -1 at the end.
2. The plus (`+`) sign is the **string concatenation** operator.
3. The asterisk (`*`) is the **repetition** operator.

```
print list + tinylis # Prints concatenated lists
['abcd', 786, 2.23, 'john', 70.200000000000003, 123, 'john']
```

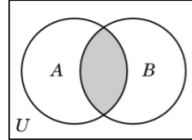
Python Tuples

1. Tuples are enclosed within **parentheses (())**
2. Lists are enclosed in brackets (`[]`) and their elements and size can be changed, while tuples are enclosed in parentheses (`()`) and **cannot be updated.**
3. Tuples can be thought of as **read-only lists**

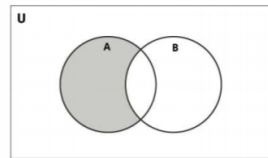
Mathematical operations

```
>>> a = set([1, 2, 3])
>>> b = set([2, 3, 4])
```

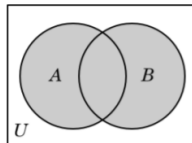
```
>>> a.intersection(b)
set([2, 3])
>>> a & b
set([2, 3])
```



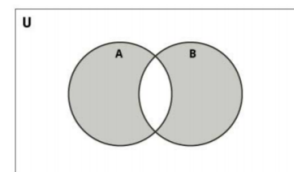
```
>>> a.difference(b)
set([1])
>>> a - b
set([1])
```



```
>>> a.union(b)
set([1, 2, 3, 4])
>>> a | b
set([1, 2, 3, 4])
```



```
>>> a.symmetric_difference(b)
set([1, 4])
>>> a ^ b
set([1, 4])
```



frozenset

1. The frozenset type is **immutable and hashable**
2. contents cannot be altered after it is created
3. It can be used as a dictionary key or as an element of another set

Python Arithmetic Operators

- Assume variable a holds 10 and variable b holds 20

Operator	Description	Example
+	Addition - Adds values on either side of the operator	a + b will give 30
-	Subtraction - Subtracts right hand operand from left hand operand	a - b will give -10
*	Multiplication - Multiplies values on either side of the operator	a * b will give 200
/	Division - Divides left hand operand by right hand operand	b / a will give 2
%	Modulus - Divides left hand operand by right hand operand and returns remainder	b % a will give 0
**	Exponent - Performs exponential (power) calculation on operators	a**b will give 10 to the power 20
//	Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed.	9//2 is equal to 4 and 9.0//2.0 is equal to 4.0

- Assume variable a holds 10 and variable b holds 20

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	c = a + b will assign value of a + b into c
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	c += a is equivalent to c = c + a
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	c -= a is equivalent to c = c - a
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	c *= a is equivalent to c = c * a
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	c /= a is equivalent to c = c / a
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	c %= a is equivalent to c = c % a
**=	Exponent AND assignment operator, Performs exponential (power) calculation on operators and assign value to the left operand	c **= a is equivalent to c = c ** a
//=	Floor Division and assigns a value, Performs floor division on operators and assign value to the left operand	c //= a is equivalent to c = c // a

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(a & b) will give 12 which is 0000 1100
	Binary OR Operator copies a bit if it exists in either operand.	(a b) will give 61 which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(a ^ b) will give 49 which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~a) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	a << 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	a >> 2 will give 15 which is 0000 1111

Control Statement	Description
break statement	Terminates the loop statement and transfers execution to the statement immediately following the loop.
continue statement	Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
pass statement	The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute.

1. **range** 是全部產生完後，**return** 一個 **list** 回來使用。
2. **xrange** 是一次產生一個值，並 **return** 一個值回來，所以 **xrange** 只適用於 **loop**。不能以 **list** 的方法搜尋

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
```


The Anonymous Functions

1. You can use the **lambda** keyword to create **small anonymous functions**.

```
#!/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 )
```

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).

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 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 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

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.