Python Identifiers

- A Python identifier is a name used to identify a variable, function, class, module, or other object.
- An identifier starts with a letter
  - A to Z
  - a to z
  - an underscore (_) followed by more letters, underscores and digits (0 to 9).
- Python does not allow punctuation characters such as @, $ and % within identifiers.
- Python is a case sensitive programming language.
  - **Manpower** and **manpower** are two different identifiers.

Python Identifiers

- Here are following identifier naming convention for Python:
  - Class names start with an uppercase letter and all other identifiers with a lowercase letter.
  - Starting an identifier with a **single leading underscore** (_) indicates by convention that the identifier is meant to be private.
    - _single_leading_underscore: weak "internal use" indicator.
  - Starting an identifier with **two leading underscores** (__) indicates a strongly private identifier.
    - a *double underscore* (__) is private; anything else isn't private.
  - If the identifier also ends with **two trailing underscores**, the identifier is a language-defined special name.

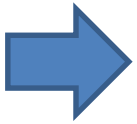
(e.g. __spirit__).

Private Variable

- In Python, although it defined the private variable, it is not as mandatory as a general programming language.
- If the outside world insists on using it, it can still be called.

Module Private and Class Private

```
# moduleA.py

def _foo():
    return 'hi'

from moduleA import *
    print _foo()
```



```
Traceback (most recent call last):
File "D:\priavte\__init__.py", line 4, in <module>
NameError: name '_foo' is not defined
```

```
from moduleA  import _foo
print _foo()
```

```
class Foo():
    __aoo = 123
    def __boo(self):
        return 123
    def coo(self):
        return 456
    f = Foo()

print dir(f)
print f._Foo__boo()
print f.__boo()
```

As imports something, all members starting with underscore will be skipped

Reserved Words

and	exec	not
assert	finally	or
break	for	pass
class	from	print
continue	global	raise
def	if	return
del	import	try
elif	in	while
else	is	with
except	lambda	yield

Lines and Indentation

- There are **no braces** "()" to indicate blocks of code for class and function definitions or flow control.
- Blocks of code are denoted by **line indentation**, which is rigidly enforced.
- The number of spaces in the indentation is variable, but all statements within the block must be indented the same amount.

```
if True:
    print "True"
else:
  print "False"
```

```
if True:
    print "Answer"
    print "True"
else:
    print "Answer"
  print "False"
```

Multi-Line Statements

- Statements in Python typically end with a new line. (without ;)
- Python allows the use of the **line continuation character (\)** to denote that the line should continue

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

Quotation in Python

- Python accepts single ('), double (") and triple (''' or """) quotes to denote string literals, as long as the same type of quote starts and ends the string.
- The triple quotes can be used to span the string across multiple lines

```
word = 'word'
sentence = "This is a sentence."
paragraph = """This is a paragraph. It is
made up of multiple lines and sentences."""
```

Comments in Python

- A hash sign (#) that is not inside a string literal begins a comment.
- All characters after the # and up to the physical line end are part of the comment and the Python interpreter ignores them.

```
#!/usr/bin/python

# First comment
print "Hello, Python!";  # second comment
```

Multiple Statements on a Single Line

- The **semicolon (;)** allows multiple statements on the single line, starting a new code block.

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

Multiple Statement Groups as Suites

- A group of individual statements, which make a single code block are called **suites** in Python.
- Compound or complex statements, such as **if**, **while**, **def**, and **class**, are those which require a header line and a suite.
- Header lines begin the statement (with the keyword) and terminate with a **colon** (**:**) and are followed by one or more lines which make up the suite.

```
if expression :
    suite
elif expression :
    suite
else :
    suite
```

Command Line Arguments

- You may have seen, for instance, that many programs can be run so that they provide you with some basic information about how they should be run.
- Python enables you to do this with -h:

```
$ python -h
usage: python [option] ... [-c cmd | -m mod | file | -] [arg] ...
Options and arguments (and corresponding environment variables):
-c cmd : program passed in as string (terminates option list)
-d     : debug output from parser (also PYTHONDEBUG=x)
-E     : ignore environment variables (such as PYTHONPATH)
-h     : print this help message and exit

[ etc. ]
```

Python Debug (pdb)

q (quit): 離開
p [some variable](print): 顯示出某個變數的值
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

```
import pdb
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()
b = [1,2,3,4,5,6,7,8,9]


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

```
ipdb> n
> c:\users\user\pdb_examp.py(19)<module>()
        17 b = [1,2,3,4,5,6,7,8,9]
        18
---->   19 for i in a:
        20     for j in b:
        21         print(complex_sum(i,j))

ipdb> n
> c:\users\user\pdb_examp.py(20)<module>()
        17 b = [1,2,3,4,5,6,7,8,9]
        18
        19 for i in a:
---->   20     for j in b:
        21         print(complex_sum(i,j))

ipdb> n
> c:\users\user\pdb_examp.py(21)<module>()
        17 b = [1,2,3,4,5,6,7,8,9]
        18
        19 for i in a:
        20     for j in b:
---->   21         print(complex_sum(i,j))

ipdb> n
do someing 1
2
> c:\users\user\pdb_examp.py(20)<module>()
        17 b = [1,2,3,4,5,6,7,8,9]
        18
        19 for i in a:
---->   20     for j in b:
        21         print(complex_sum(i,j))

ipdb> l
        15 a = [0,1,2,3,4,5,6,7,8]
        16 pdb.set_trace()
        17 b = [1,2,3,4,5,6,7,8,9]
        18
        19 for i in a:
---->   20     for j in b:
        21         print(complex_sum(i,j))

ipdb> p b
[1, 2, 3, 4, 5, 6, 7, 8, 9]

ipdb>
```

```
ipdb> n
do something 1
1
> c:\users\user\python debug.py(23)<module>(
     21
     22 for i in a:
---> 23     for j in b:
     24         print (complex_sum(i, j))
     25


ipdb> l
     18 pdb.set_trace()
     19 b = [1,2,3,4,5,6,7,8,9]
     20
     21
     22 for i in a:
---> 23     for j in b:
     24         print (complex_sum(i, j))
     25
```

Python print

Python 2

Python 3

- The simplest way to produce output is using the *print* or *print()* statement where you can pass zero or more expressions separated by commas.

print(count, miles, name)

- This function converts the expressions you pass into a string and writes the result to standard output.

Python 2

```
#!/usr/bin/python

print "Python is really a great language,", "isn't it?"
```

```
Python is really a great language, isn't it?
```

Python 3

```
print("Hello World")
```

```
Hello World
```

Assigning Values to Variables

- Python variables do not have to be explicitly declared to reserve memory space.
- The declaration happens automatically when you assign a value to a variable.
  - The equal sign (=) is used to assign values to variables.
- The operand to the left of the "= operator" is the name of the variable and the operand to the right of the "= operator" is the value stored in the variable.

```
count = 100
miles = 1000.0
name = "John"

print(count)
print(miles)
print(name)
```

```
100
1000.0
John
```

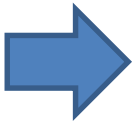
Variable

```
# assign 4 to the variable x
x = 4
```

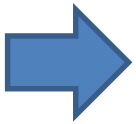
```
// C code
int x = 4;
```

```
x = 1          # x is an integer
x = 'hello'    # now x is a string
x = [1, 2, 3] # now x is a list
```

```
x = [1, 2, 3]
y = x
```



```
print(y)
```



```
[1, 2, 3]
```

Python types

- Numeric type
  - int : 42 may be transparently expanded to long through 438324932l
  - long : long int
  - float : 2.171892
  - complex : 4 + 3j
  - bool : True of False

long: long integers of non-limited length; exists only in Python 2.x

```
~$ python
Python 2.7.3
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> type(1)      # 1 是什麼型態？
<type 'int'>
>>> type(1L)     # 加上 L 呢？
<type 'long'>
>>> type(111111111111111111111111111111111) # 太長的整數會自動使用 long 型態
<type 'long'>
>>> type(3.14)   # 浮點數是 float 型態
<type 'float'>
>>> type(True)   # 布林值是 bool 型態
<type 'bool'>
>>> type(3 + 4j) # 支援複數的 complex 型態
<type 'complex'>
>>> 2 ** 100     # 2 的 100 次方
1267650600228229401496703205376L
```

```
In [7]: type
(1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111)
Out[7]: int
```

Multiple Assignment

- Python allows you to assign a single value to several variables simultaneously.

```
a = b = c = 1
```

```
a, b, c = 1, 2, "john"
```

Standard Data Types

- Python has five 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.)

```
var1 = 1
var2 = 10
```

Python types

- Str – “Hello”
- List – [69, 6.9, ‘mystring’, True]
- Tuple – (69, 6.9, ‘mystring’, True) =>immutable
- Dictionary or hash – {‘key 1’: 6.9, ‘key2’: False} - group of key and value pairs
- Set/frozenset
  - set([69, 6.9, ‘str’, True])
  - frozenset([69, 6.9, ‘str’, True]) => immutable –no duplicates & unordered

Python Strings

- 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.
- The plus (+) sign is the string concatenation operator.
- The asterisk (*) is the repetition operator.

```
#!/usr/bin/python

str = 'Hello World!'

print str          # Prints complete string
print str[0]       # Prints first character of the string
print str[2:5]     # Prints characters starting from 3rd to 5th
print str[2:]      # Prints string starting from 3rd character
print str * 2      # Prints string two times
print str + "TEST" # Prints concatenated string
```

```
Hello World!
H
llo
llo World!
Hello World!Hello World!
Hello World!TEST
```

Python Lists

- A list contains items separated by commas (**,**) and enclosed within square brackets (**[]**).
- To some extent, lists are similar to arrays in C.
  - One difference, a list can be of different data types.
- The values stored in a list can be accessed using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the list and working their way to end -1.
- The plus (+) sign is the list concatenation operator, and the asterisk (*) is the repetition operator.

Python Lists

```
list = ['abcd',786,2.23,'john',70.2]
tinylist = [123,'john']
print (list)
print (list[0])
print (list[1:3])
print (list[2:])
print (tinylist *2)
print (list + tinylist)
```

```
['abcd', 786, 2.23, 'john', 70.2]
abcd
[786, 2.23]
[2.23, 'john', 70.2]
[123, 'john', 123, 'john']
['abcd', 786, 2.23, 'john', 70.2, 123, 'john']
```

Python Tuples

- A tuple consists of a number of values separated by commas .
- Tuples are enclosed within parentheses (**( )**).
- The main differences between lists and tuples are:
  - Lists are enclosed in brackets (**[ ]**) and their elements and size can be changed, while tuples are enclosed in parentheses (**( )**) and cannot be updated.
  - Tuples can be thought of as **read-only** lists

```
#!/usr/bin/python

tuple = ( 'abcd', 786 , 2.23, 'john', 70.2  )
tinytuple = (123, 'john')

print tuple           # Prints complete list
print tuple[0]        # Prints first element of the list
print tuple[1:3]      # Prints elements starting from 2nd till 3rd
print tuple[2:]       # Prints elements starting from 3rd element
print tinytuple * 2   # Prints list two times
print tuple + tinytuple # Prints concatenated lists
```

```
('abcd', 786, 2.23, 'john', 70.200000000000003)
abcd
(786, 2.23)
(2.23, 'john', 70.200000000000003)
(123, 'john', 123, 'john')
('abcd', 786, 2.23, 'john', 70.200000000000003, 123, 'john')
```

Tuple Example

```
#!/usr/bin/python

tuple = ( 'abcd', 786 , 2.23, 'john', 70.2  )
list = [ 'abcd', 786 , 2.23, 'john', 70.2  ]
tuple[2] = 1000    # Invalid syntax with tuple
list[2] = 1000     # Valid syntax with list
```

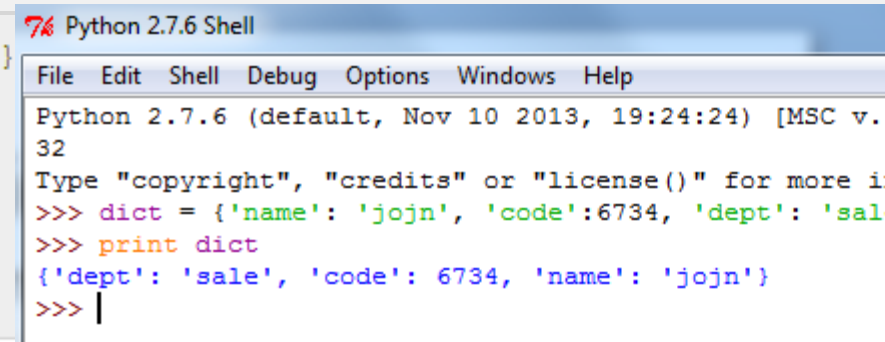
```
>>> t = ([1, 2], [3, 4])
>>> t
([1, 2], [3, 4])
>>> t[0] = [10, 20]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Python Dictionary

- A dictionary key can be almost any Python type, but are usually numbers or strings.
  - Values, on the other hand, can be any arbitrary Python object.
- Dictionaries are enclosed by curly braces ({ }) and values can be assigned and accessed using square braces ([])

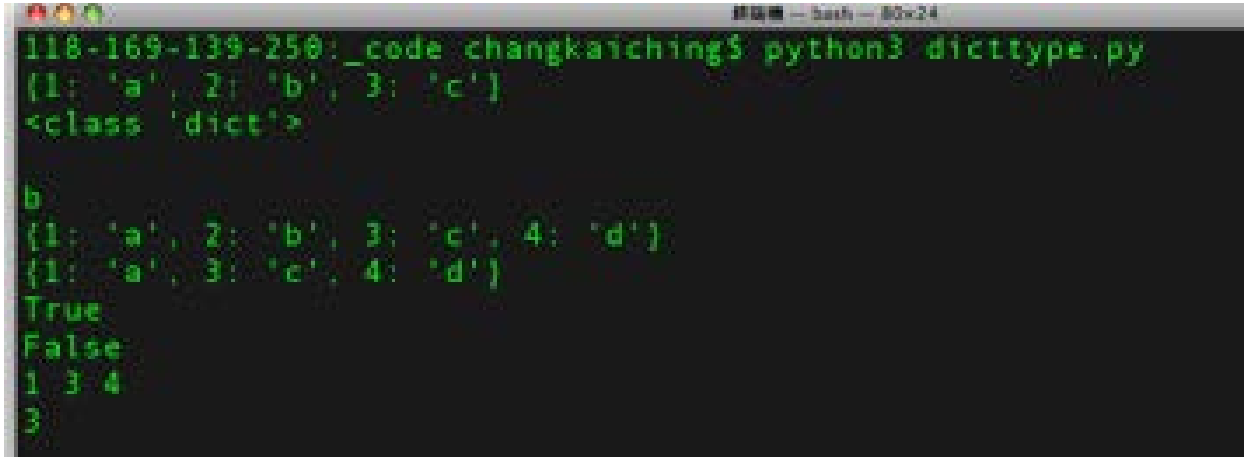
```
This is one
This is two
{'dept': 'sales', 'code': 6734, 'name': 'john'}
['dept', 'code', 'name']
['sales', 6734, 'john']
```

```
#!/usr/bin/python

dict = {}
dict['one'] = "This is one"
dict[2]     = "This is two"

tinydict = {'name': 'john','code':6734, 'dept': 'sales'}


print dict['one']       # Prints value for 'one' key
print dict[2]           # Prints value for 2 key
print tinydict          # Prints complete dictionary
print tinydict.keys()   # Prints all the keys
print tinydict.values() # Prints all the values
```



Dictionary – Python 3.x

```
d = {1:"a", 2:"b", 3:"c"}
print(d)
print(type(d))
print()

print(d[2])
d[4] = "d"
print(d)
del d[2]
print(d)
print(3 in d)
print(3 not in d)
for i in iter(d):
    print(i, end=" ")
print()
print(len(d))
print()
```




```
iter(object[, sentinel])
```

Nest Dictionary

- Dictionary can be used as a tiny database.

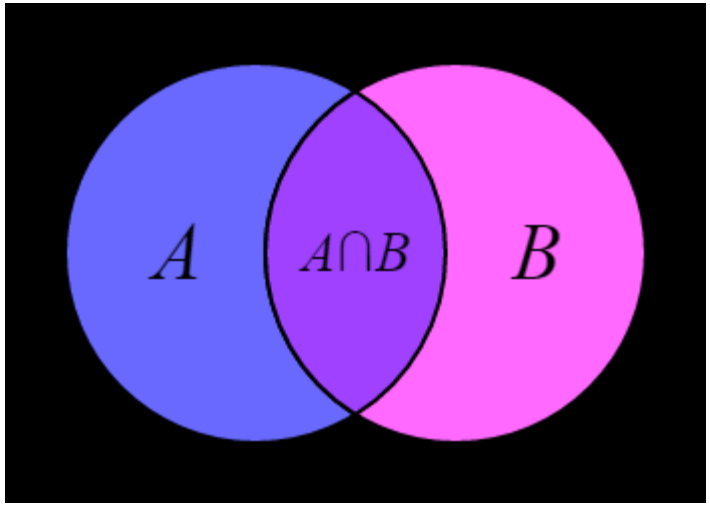
```
people = {
    'Alice': {
        'phone': '2341',
        'addr': 'Foo drive 23'},
    'Beth': {
        'phone': '9102',
        'addr': 'Bar street 42'},
    'Cecil': {
        'phone': '3158',
        'addr': 'Baz avenue 90'}
}
```

```
>>> people['Beth']['phone']
'9102'
>>> people['Alice']['addr']
'Foo drive 23'
```

Introduction to Sets

- A set is an unordered collection with no duplicate elements.
- It is a computer implementation of the mathematical concept of a finite set.
- Set creation:

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



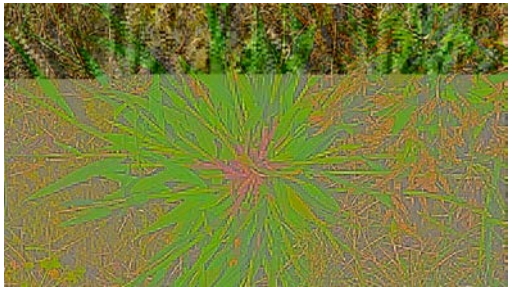
Set

- Checking membership
- Removing duplicates

```
fruits = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
basket = set(fruits)
print(basket)
```

➔set(['orange', 'pear', 'apple', 'banana'])

```
>>> 'orange' in basket
True
>>> 'crabgrass' in basket
False
```



Set Methods

- add
- clear
- copy
- difference
- difference_update
- discard
- intersection
- intersection_update
- isdisjoint
- issubset
- issuperset

https://docs.python.org/3/library/stdtypes.html

Modifying & Membership

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

- Checking for Membership
- Return the Boolean value

```
>>> c = a & b
>>> c = set([2, 3])
>>> c.issubset(a)
True
>>> a.issuperset(c)
True
```

Set modifying

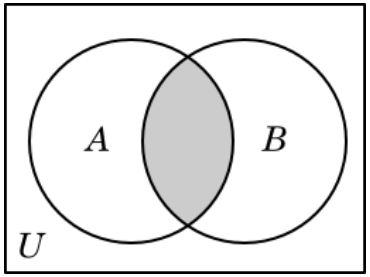
- in place

```
>>> a.add(4)
>>> a
set([1, 2, 3, 4])
>>> a.remove(1)
>>> a
set([2, 3, 4])
>>> a.clear()
>>> a
set([])
>>> a.update(b)
>>> a
set([2, 3, 4])
```

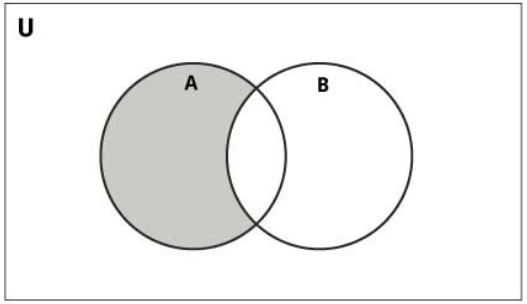
Mathematical operations

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

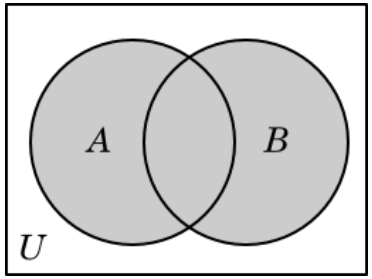
```
>>> a.intersection(b)
set([2, 3])
>>> a & b
set([2, 3])
```



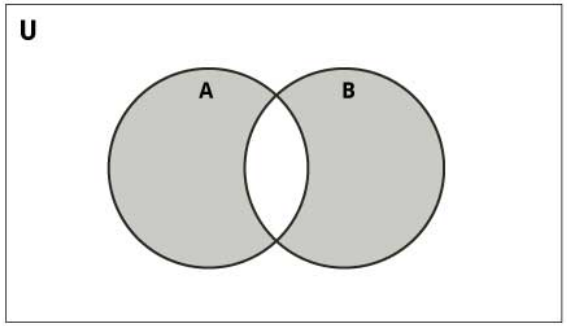

```
>>> a.difference(b)
set([1])
>>> a - b
set([1])
```




```
>>> 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

- The frozenset type is immutable and hashable
  - Its contents cannot be altered after it is created
  - It can be used as a dictionary key or as an element of another set

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

```
>>> a.add(b)
Traceback (most recent call last): File "", line 1, in
TypeError: unhashable type: 'set' >>>
a.add(frozenset(b))
>>> a
set([1, 2, 3, frozenset([2, 3, 4])])
```

```
File "C:/Users/USER/set.py", line 12, in <module>
  a.add(b)

TypeError: unhashable type: 'set'
```

Function	Description
int(x [,base])	Converts x to an integer. base specifies the base if x is a string.
long(x [,base])	Converts x to a long integer. base specifies the base if x is a string.
float(x)	Converts x to a floating-point number.
complex(real [,imag])	Creates a complex number.
str(x)	Converts object x to a string representation.
repr(x)	Converts object x to an expression string.
eval(str)	Evaluates a string and returns an object.
tuple(s)	Converts s to a tuple.
list(s)	Converts s to a list.
set(s)	Converts s to a set.
dict(d)	Creates a dictionary. d must be a sequence of (key,value) tuples.
frozenset(s)	Converts s to a frozen set.
chr(x)	Converts an integer to a character.

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

Example

- Python 2.7

```
>>> 10 / 3
3
>>> 10 // 3
3
>>> 10 / 3.0
3.3333333333333335
>>> 10 // 3.0
3.0
>>>
```

Python 3.7

```
In [16]: 10/3
Out[16]: 3

In [17]: 10//3
Out[17]: 3

In [18]: 10/3.0
Out[18]: 3.3333333333333335

In [19]: 10//3.0
Out[19]: 3.0
```

Python Comparison Operators

- Assume variable a holds 10 and variable b holds 20

Operator	Description	Example
==	Checks if the value of two operands are equal or not, if yes then condition becomes true.	(a == b) is not true.
!=	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.	(a != b) is true.
<>	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.	(a <> b) is true. This is similar to != operator.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(a > b) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(a < b) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(a >= b) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(a <= b) is true.

Python Assignment Operators

- 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 assigne 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 Dividion and assigns a value, Performs floor division on operators and assign value to the left operand	c //= a is equivalent to c = c // a

Python Bitwise Operators

- Assume if a = 60; and b = 13;
- Now in binary format, they will be as follows:
- a = 0011 1100;
- b = 0000 1101
- a&b = 0000 1100
- a|b = 0011 1101
- a^b = 0011 0001
- ~a = 1100 0011

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 eather 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 efect 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

Python Logical Operators

- Assume variable a holds 10 and variable b holds 20

Operator	Description	Example
and	Called Logical AND operator. If both the operands are true then then condition becomes true.	(a and b) is true.
or	Called Logical OR Operator. If any of the two operands are non zero then then condition becomes true.	(a or b) is true.
not	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	not(a and b) is false.

Python Membership Operators

- Python has membership operators, which test for membership in a sequence, such as strings, lists, or tuples

Operator	Description	Example
in	Evaluates to true if it finds a variable in the specified sequence and false otherwise.	x in y, here **in** results in a 1 if x is a member of sequence y.
not in	Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.	x not in y, here **not in** results in a 1 if x is not a member of sequence y.

Example

```
#!/usr/bin/python

a = 10
b = 20
list = [1, 2, 3, 4, 5 ];

if ( a in list ):
   print "Line 1 - a is available in the given list"
else:
   print "Line 1 - a is not available in the given list"

if ( b not in list ):
   print "Line 2 - b is not available in the given list"
else:
   print "Line 2 - b is available in the given list"

a = 2
if ( a in list ):
   print "Line 3 - a is available in the given list"
else:
   print "Line 3 - a is not available in the given list"
```

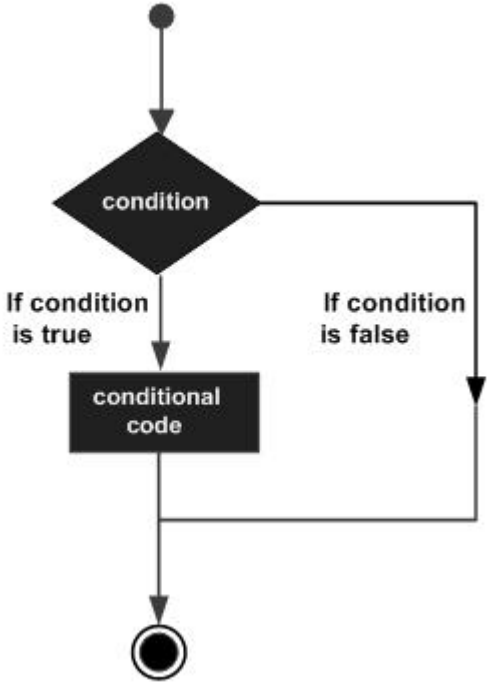
```
Line 1 - a is not available in the given list
Line 2 - b is not available in the given list
Line 3 - a is available in the given list
```

Python Operators Precedence

Operator	Description
**	Exponentiation (raise to the power)
~ + -	Ccomplement, unary plus and minus (method names for the last two are +@ and -@)
* / % //	Multiply, divide, modulo and floor division
+ -	Addition and subtraction
>> <<	Right and left bitwise shift
&	Bitwise 'AND'
^ |	Bitwise exclusive `OR' and regular `OR'
<= < > >=	Comparison operators
<> == !=	Equality operators
= %= /= //= -= += *= **=	Assignment operators
is is not	Identity operators
in not in	Membership operators
not or and	Logical operators

Python Decision Making

Statement	Description
if statements	An **if statement** consists of a boolean expression followed by one or more statements.
if...else statements	An **if statement** can be followed by an optional **else statement**, which executes when the boolean expression is false.
nested if statements	You can use one **if** or **else if** statement inside another **if** or **else if** statement(s).




```
#!/usr/bin/python

var = 100

if ( var  == 100 ) : print "Value of expression is 100"

print "Good bye!"
```

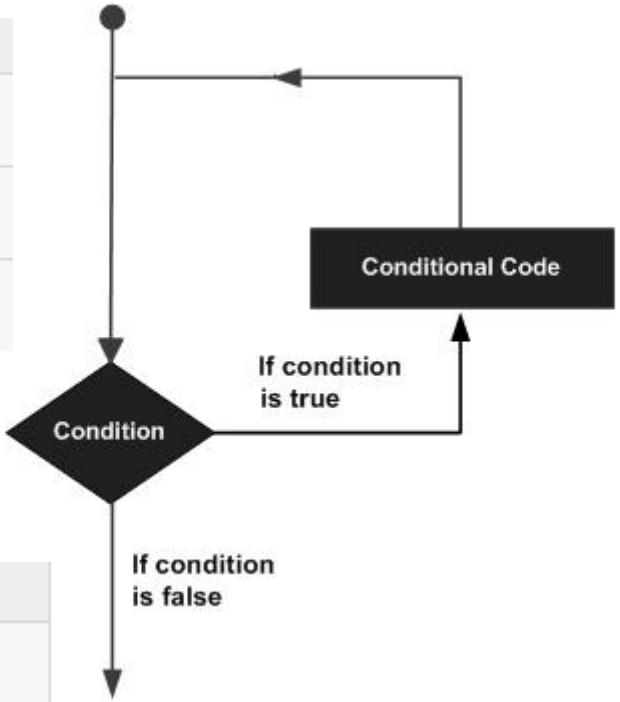
```
Value of expression is 100
Good bye!
```

Conditionals Cont.

- **if** (value **is not None**) **and** (value == 1):
  print (value equals 1)
  print (more can come in this block)
- **if** (list1 <= list2) **and** (**not** age < 80):
  print (1 = 1, 2 = 2, but 3 <= 7 so its True)
- **if** (job == "millionaire") **or** (state != "dead"):
  print (a suitable husband found)
  else:
  print (not suitable)
- **if** ok: **print** (ok)

Python Loops

Loop Type	Description
while loop	Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
for loop	Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
nested loops	You can use one or more loop inside any another while, for or do..while loop.



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.

Loops/Iterations

- sentence = ['Marry','had','a','little','lamb']
  for word in sentence:
      print (word, len(word))

```
Marry 5
had 3
a 1
little 6
lamb 4
```

- for i in range(10):
      print (i)
  for i in range(1000): # does not allocate all initially
      print (i)
- while True:
      pass
- for i in range(10):
      if i == 3: continue
      if i == 5: break
      print (i)

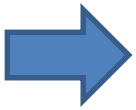
```
0
1
2
4
```

pass

- while 1:
  ... pass # Busy-wait for keyboard interrupt
  ...
- class MyEmptyClass:
  ... pass
  ...

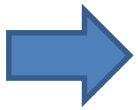
Example

```
for i in "Hi Python":
    if i == "t":
        break
    print(i)
```



```
H
i

P
y
```

```
sequences = [0, 1, 2, 3, 4, 5]
i = 0
while 1: #判斷條件值為1, 代表迴圈永遠成立
    print(sequences[i], end = " ")
    i = i + 1
    if i == len(sequences):
        print()
        print("No elements left.")
        break
```



```
0 1 2 3 4 5
No elements left.
```

range() and xrange()

- range() can construct a numeral list
  - range(start, stop, step)

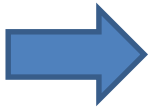
```
>>> range(5)
[0, 1, 2, 3, 4]
>>> range(1,5)
[1, 2, 3, 4]
>>> range(0,6,2)
[0, 2, 4]
```

- xrange() return a generator

```
>>> xrange(5)
xrange(5)
>>> list(xrange(5))
[0, 1, 2, 3, 4]
>>> xrange(1,5)
xrange(1, 5)
>>> list(xrange(1,5))
[1, 2, 3, 4]
>>> xrange(0,6,2)
xrange(0, 6, 2)
>>> list(xrange(0,6,2))
[0, 2, 4]
```

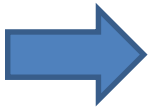
range()

```
a = range(0,100)
print(type(a))
print (a)
print(a[0],a[1])
```



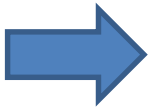
```
<class 'range'>
range(0, 100)
0 1
```

```
for i in range(10):
    print(i, end=" ")

print() #換行
for i in range(20, 2, -2):
    print(i, end=" ")
```



```
0 1 2 3 4 5 6 7 8 9
20 18 16 14 12 10 8 6 4
```

```
for i in range(1, 10):
    for j in range(1, 10):
        if j == 9:
            print("\t", i*j) # j == 9時，換行
        else:
            print("\t", i*j, end = '') # j < 9時，不換行
```



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

Difference of range() and xrange()

- xrange()

```
1. a = xrange(0,100)
2. print type(a)
3. print a
4. print a[0], a[1]
```

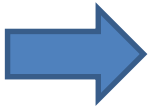
```
1. <type 'xrange'>
2. xrange(100)
3. 0 1
```

```
    a = xrange(0,100)

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

xrange: exists only in Python 2.x

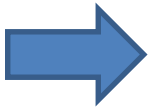
Example

```
x = int(input("please input a number:"))
y = int(input("please input b number:"))
if x > y :
    c = x
else:
    c = y
for k in range(2, c):
    if x % k == 0 and y % k == 0:
        print(k)
```



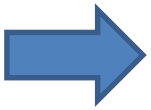
```
please input a number:10

please input b number:20
2
5
10
```

```
x = int(input("please input a number:"))
y = int(input("please input b number:"))
if x > y :
    c = x
else:
    c = y
for k in range(c, 2, -1):
    if x % k == 0 and y % k == 0:
        print(k)
        break
```



```
please input a number:10

please input b number:20
10
```

```python
x = int(input("please input a number:"))
y = int(input("please input b number:"))
if x > y :
    c = x
else:
    c = y
m = (x + 1) * (y + 1)
for k in range(c, m):
    if k % x == 0 and k % y == 0:
        print(k)
        break
```



```
please input a number:20

please input b number:10
20
```

List Methods

- list.**append**(*x*)
- list.**extend**(*iterable*)
- list.**insert**(*i*, *x*)
- list.**remove**(*x*)
- list.**pop**([*i*])
- list.**count**(*x*)
- list.**reverse**()
- list.**copy**()
- list.**index**(*x*[, *start*[, *end*]])
- list.**clear**()

List

```
fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
print(fruits.count('apple'))
print(fruits.index('banana'))
print(fruits.index('banana', 4))
fruits.reverse()
print(fruits)
fruits.append('grape')
print(fruits)
fruits.sort()
print(fruits)
print(fruits.pop())
```

```
2
3
6
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange']
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange', 'grape']
['apple', 'apple', 'banana', 'banana', 'grape', 'kiwi', 'orange', 'pear']
pear
```

List Applied to Stack

```
In [23]: stack = [3, 4, 5]

In [24]: stack.append(6)

In [25]: stack.append(7)

In [26]: stack
Out[26]: [3, 4, 5, 6, 7]

In [26]:

In [27]: stack.pop()
Out[27]: 7

In [28]: stack
Out[28]: [3, 4, 5, 6]

In [29]: stack.pop()
Out[29]: 6

In [30]: stack.pop()
Out[30]: 5

In [31]: stack
Out[31]: [3, 4]
```

Example

```
knights = {'gallahad': 'the pure', 'robin': 'the brave'}
for k, v in knights.items():
    print(k, v)
```

```
gallahad the pure
robin the brave
```

```
questions = ['name', 'quest', 'favorite color']
answers = ['lancelot', 'the holy grail', 'blue']
for q, a in zip(questions, answers):
    print('What is your {0}?  It is {1}.'.format(q, a))
```

```
What is your name?  It is lancelot.
What is your quest?  It is the holy grail.
What is your favorite color?  It is blue.
```

`zip()`：它可以同時迭代多個 list

Python Exceptions Handling

- Python provides two very important features to handle any unexpected error and to add debugging capabilities in them.
  - **Assertions**
  - **Exception Handling**

Assertions in Python

- An assertion is a sanity-check that you can turn on or turn off when you are done with your testing of the program.
- The easiest way to think of an assertion is to liken it to a **raise-if** statement (or to be more accurate, a raise-if-not statement).
- An expression is tested, and if the result comes up false, an exception is raised.

```
assert Expression[, Arguments]
```

Example

```
def kelvinToFahrenheit(Temperature):
    assert(Temperature >= 0), "Colder than absolute zero!!!"
    return ((Temperature-273)*1.8)+32

print(kelvinToFahrenheit(273))
print(int(kelvinToFahrenheit(505.78)))
print(kelvinToFahrenheit(-5))
```

```
32.0
451
Traceback (most recent call last):
  File "C:\Users\user\untitled0.py", line 32, in <module>
    print(kelvinToFahrenheit(-5))

  File "C:\Users\user\untitled0.py", line 27, in kelvinToFahrenheit
    assert(Temperature >= 0), "Colder than absolute zero!!!"

AssertionError: Colder than absolute zero!!!
```

What is Exception?

- An exception is an event, which is the execution of a program that disrupts the normal flow of the program's instructions.
  - a Python script encounters a situation that it cannot cope with, and it raises an exception.
- An exception is a Python object that represents an error.
- When a Python script raises an exception, it must either handle the exception immediately otherwise it terminates and quits.

Handling an Exception

- If you have some *suspicious code* that may raise an exception, you can defend your program by placing the suspicious code in a try block.
- After the try block, include an except statement, followed by a block of code which handles the problem as elegantly as possible.

```
try:
   You do your operations here;
   ......................
except ExceptionI:
   If there is ExceptionI, then execute this block.
except ExceptionII:
   If there is ExceptionII, then execute this block.
   ......................
else:
   If there is no exception then execute this block.
```

Important Points

- Few important points about the above-mentioned syntax
  - A single try statement can have multiple except statements.
    - The try block contains statements that may *throw* different types of exceptions.
  - You also provide a generic except clause, which handles any exception.
  - After the except clause(s), you can include an else-clause.
  - The code in the else-block executes if the code in the try block does not raise an exception.
    - The else-block is a good place for code that does not need the try block's protection.

Example

- This example opens a file, writes content in the file and comes out gracefully because there is no problem at all

```
#!/usr/bin/python

try:
   fh = open("testfile", "w")
   fh.write("This is my test file for exception handling!!")
except IOError:
   print "Error: can\'t find file or read data"
else:
   print "Written content in the file successfully"
   fh.close()
```

```
Written content in the file successfully
```

This example tries to open a file where you do not have read permission, so it raises an exception

```
#!/usr/bin/python

try:
   fh = open("testfile", "r")
   fh.write("This is my test file for exception handling!
except IOError:
   print "Error: can\'t find file or read data"
else:
   print "Written content in the file successfully"
```

```
Error: can't find file or read data
```

```python
try:
    a = int(input('輸入 0～9：'))
    if a>9:          # 如果輸入的 a 大於 9
        raise        # 強制中斷，拋出錯誤資訊席
    print(a)
except :
    print('有錯誤喔～')    # 收到錯誤訊息，顯示錯誤
```

```
輸入 0～9：5
5
```

```
輸入 0～9：11
有錯誤喔～
```

```python
try:
    a = int(input('輸入 0～9：'))
    if a>10:
        raise ValueError('數字不在範圍內')
    print(a)
except ValueError as msg:     # 如果輸入範圍外的數字，執行這邊的程式
    print(msg)
except :                      # 如果輸入的不是數字，執行這邊的程式
    print('有錯誤喔～')
print('繼續執行')
```

```
輸入 0～9：5
5
繼續執行
```

```
輸入 0～9：11
數字不在範圍內
繼續執行
```

```python
try:
    a = int(input('輸入 0～9：'))
    if a>10:
        assert False, '數字不在範圍內'
    print(a)
except AssertionError as msg:
    print(msg)
except :
    print('有錯誤喔～')
print('繼續執行')
```

```
輸入 0～9：5
5
繼續執行
```

```
輸入 0～9：11
數字不在範圍內
繼續執行
```

```
輸入 0～9：xyz
有錯誤喔～
繼續執行
```

The try-finally Clause

```
#!/usr/bin/python

try:
   fh = open("testfile", "w")
   fh.write("This is my test file for exception handling!!")
finally:
   print "Error: can\'t find file or read data"
```

```
Error: can't find file or read data
```

- Same example can be written more cleanly as follows

```
#!/usr/bin/python

try:
   fh = open("testfile", "w")
   try:
      fh.write("This is my test file for exception handling!!")
   finally:
      print "Going to close the file"
      fh.close()
except IOError:
   print "Error: can\'t find file or read data"
```

Argument of an Exception

- An exception can have an *argument*, which is a value that gives additional information about the problem.
  - The contents of the argument vary by exception.
- You capture an exception's argument by supplying a variable in the except clause as follows

```
try:
   You do your operations here;
   ......................
except ExceptionType as Argument:
   You can print value of Argument here...
```

Example

```
def temp_covert(var):
    try:
        print("The argument is a integer\n")
        return int(var)
    except ValueError as Argument:
        print("The argument does not contain number\n", Argument)


temp_covert(100)
temp_covert("abc")
```

```
The argument is a integer

The argument is a integer

The argument does not contain number
 invalid literal for int() with base 10: 'abc'
```

User-Defined Exceptions

- Python also allows you to create your own exceptions by deriving classes from the standard built-in exceptions.
- Here, a class is created that is subclassed from *RuntimeError*.
  - when you need to display more specific information
  - when an exception is caught.

```
class Networkerror(RuntimeError):
    def __init__(self,args):
        self.args =args

try:
    raise Networkerror("Bad hostname")
except Networkerror as e:
    print("error message",e.args)
```

```
error message ('B', 'a', 'd', ' ', 'h', 'o', 's', 't', 'n', 'a', 'm', 'e')
```

```
In [1]: import builtins

In [2]: dir(builtins)
Out[2]:
['ArithmeticError',
 'AssertionError',
 'AttributeError',
 'BaseException',
 'BlockingIOError',
 'BrokenPipeError',
 'BufferError',
 'BytesWarning',
 'ChildProcessError',
 'ConnectionAbortedError',
 'ConnectionError',
 'ConnectionRefusedError',
 'ConnectionResetError',
 'DeprecationWarning',
 'EOFError',
 'Ellipsis',
 'EnvironmentError',
 'Exception',
 'False',
```

```
>>> import builtins
>>> dir(builtins)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'Buffer
Error', 'BytesWarning', 'DeprecationWarning', 'EOFError', 'Ellipsis', 'Environme
ntError', 'Exception', 'False', 'FloatingPointError', 'FutureWarning', 'Generato
rExit', 'IOError', 'ImportError', 'ImportWarning', 'IndentationError', 'IndexErr
or', 'KeyError', 'KeyboardInterrupt', 'LookupError', 'MemoryError', 'NameError',
 'None', 'NotImplemented', 'NotImplementedError', 'OSError', 'OverflowError', 'P
endingDeprecationWarning', 'ReferenceError', 'RuntimeError', 'RuntimeWarning', '
StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit', 'Ta
bError', 'True', 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError', 'Unicod
eEncodeError', 'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserW
arning', 'ValueError', 'Warning', 'WindowsError', 'ZeroDivisionError', 略...]
```

錯誤資訊	說明
NameError	使用沒有被定義的對象
IndexError	索引值超過了序列的大小
TypeError	數據類型（type）錯誤
SyntaxError	Python 語法規則錯誤
ValueError	傳入值錯誤
KeyboardInterrupt	當程式被手動強制中止
AssertionError	程式 asset 後面的條件不成立
KeyError	鍵發生錯誤
ZeroDivisionError	除以 0
AttributeError	使用不存在的屬性
IndentationError	Python 語法錯誤（沒有對齊）
IOError	Input/output異常
UnboundLocalError	區域變數和全域變數發生重複或錯誤