Python Identifiers

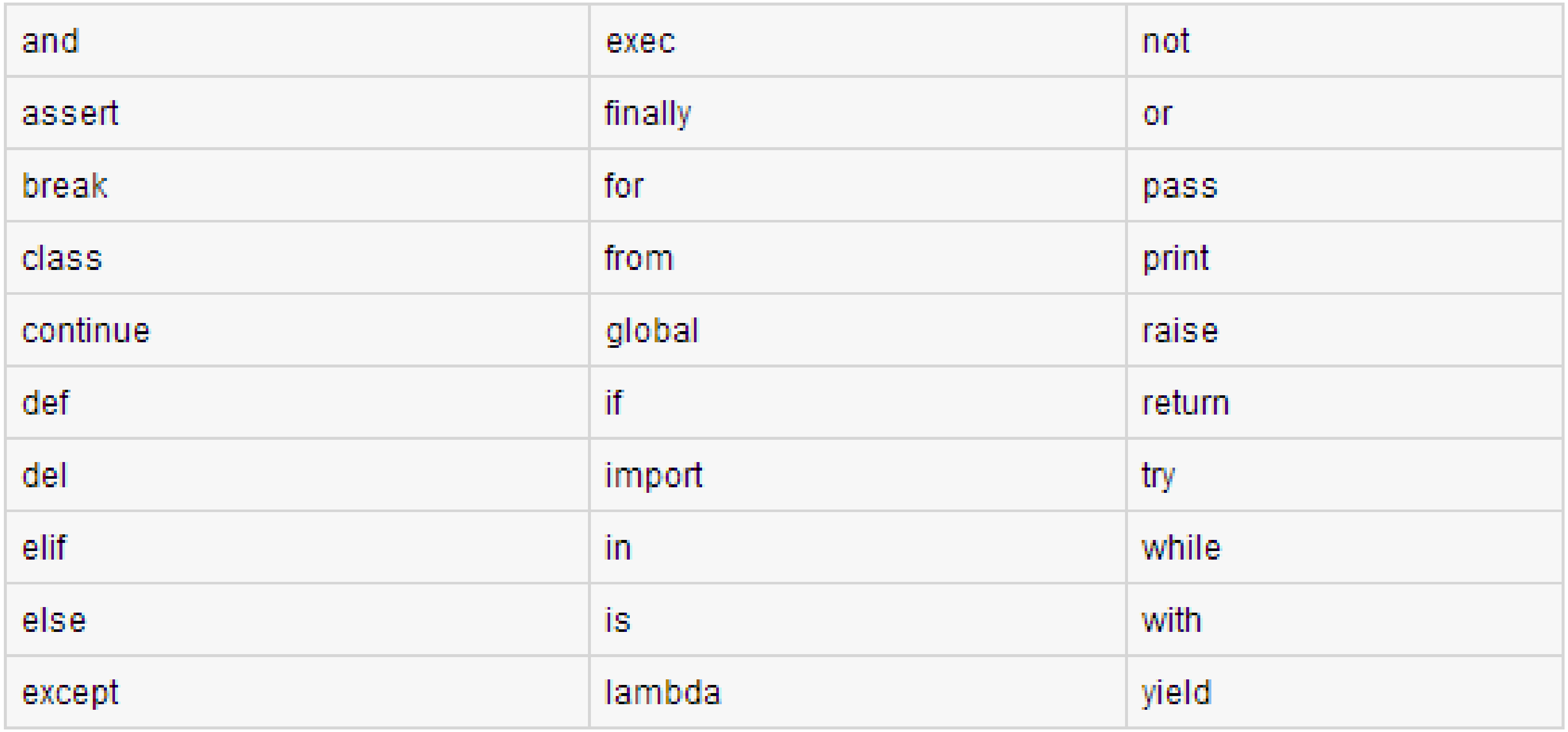
- A Python identifier is a name used to identify a variable, function, class, module or other object.
- An identifier starts with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores and digits (0 to 9).
- Python does not allow punctuation characters such as @,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

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__).

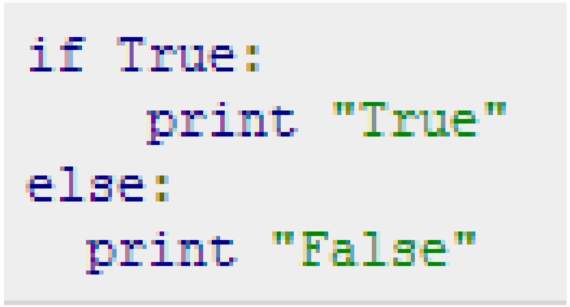
Reserved Words



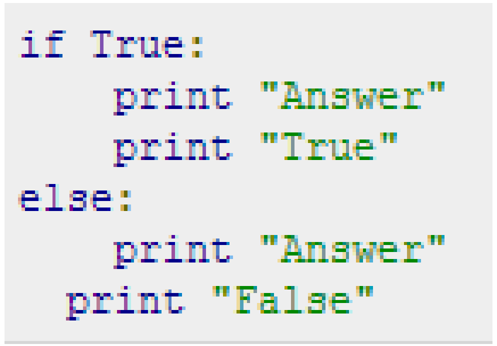
and	exec	not
assert	finally	or
break	for	pass
class	from	print
continue	global	raise
def	if	return
del	import	try
elif	in	while
else	is	with
except	lambda	yield

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.
- 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
```

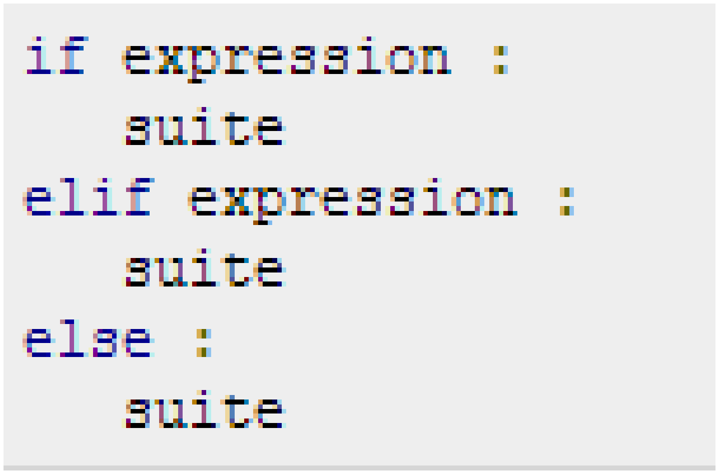
Multiple Statements on a Single Line

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

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

Multiple Statement Groups as Suites

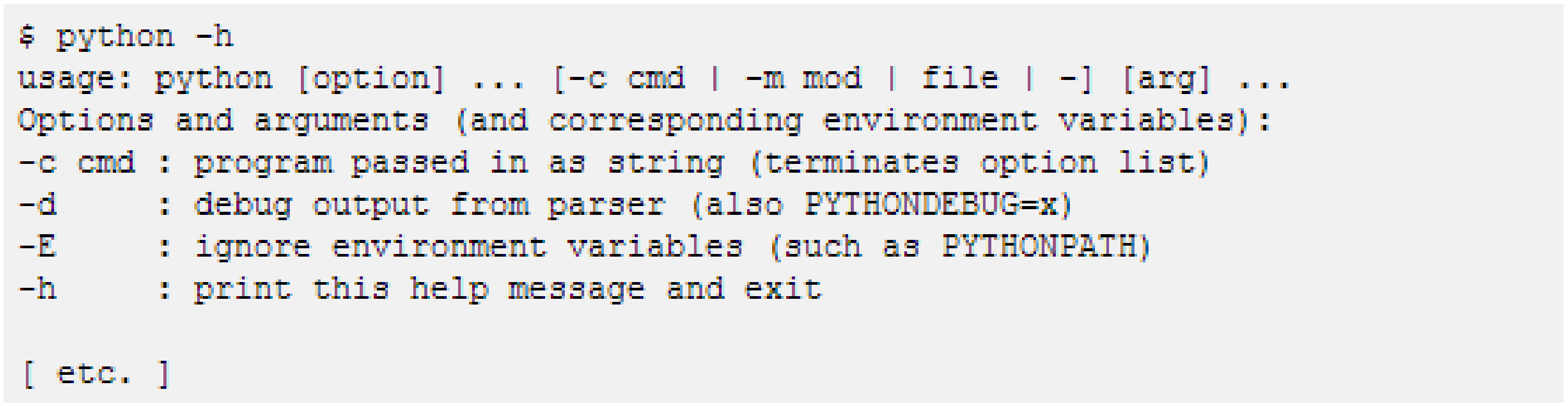
- A group of individual statements, which make a single code block are called **suites** in Python.
- Compound or complex statements, such as **if**, **while**, **def**, and **class**, are those which require a header line and a suite.
- Header lines begin the statement (with the keyword) and terminate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



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

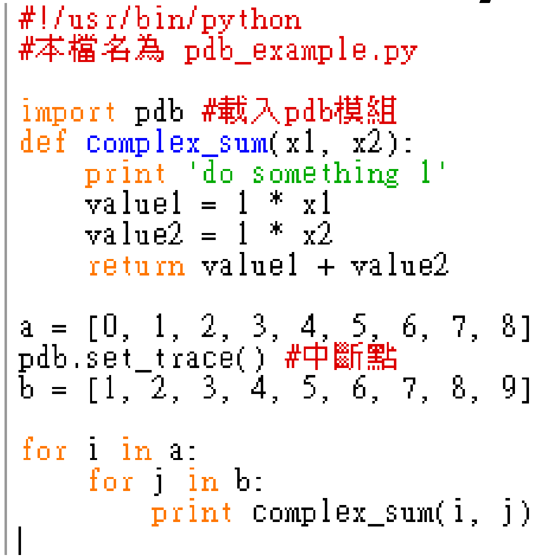
Command Line Arguments

- 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:

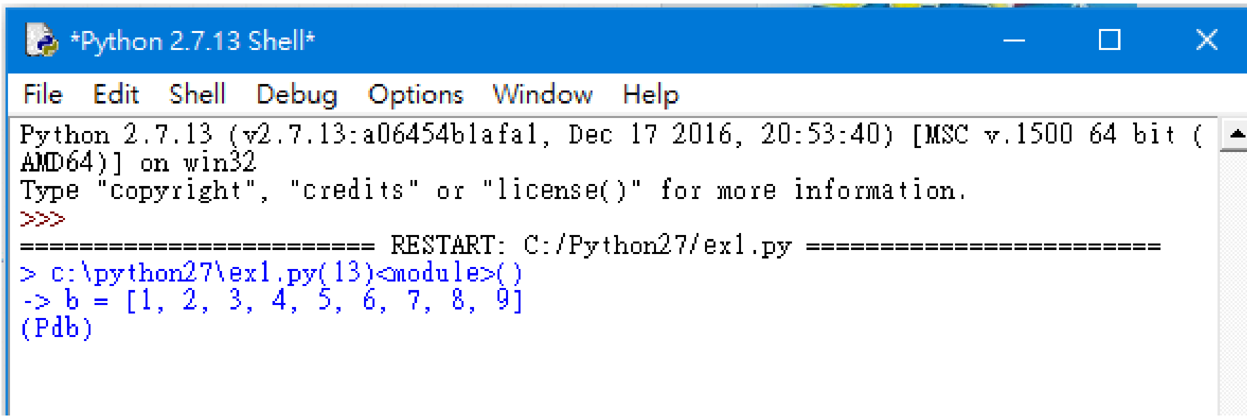


```
$ python -h
usage: python [option] ... [-c cmd | -m mod | file | -] [arg] ...
Options and arguments (and corresponding environment variables):
-c cmd : program passed in as string (terminates option list)
-d     : debug output from parser (also PYTHONDEBUG=x)
-E     : ignore environment variables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

[ etc. ]
```

Python Debug (pdb)



```
#!/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)
```

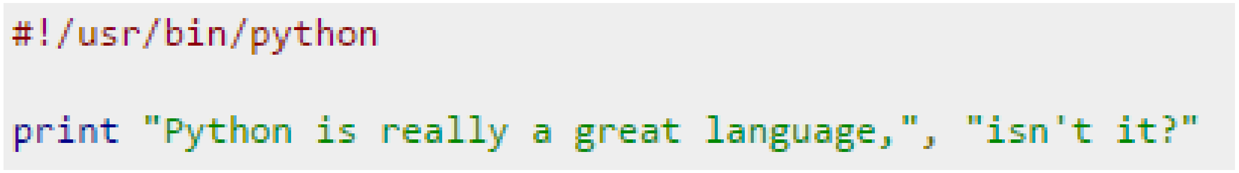


```
Python 2.7.13 (v2.7.13:a06454b1afa1, Dec 17 2016, 20:53:40) [MSC v.1500 64 bit (
AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
========================= RESTART: C:/Python27/ex1.py =========================
> c:\python27\ex1.py(13)<module>()
-> b = [1, 2, 3, 4, 5, 6, 7, 8, 9]
(Pdb)
```

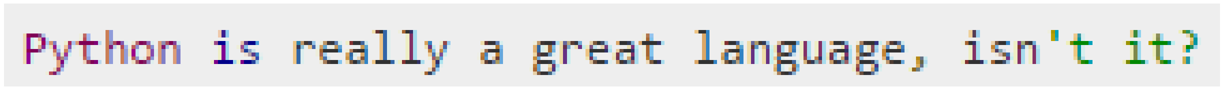
q(quit): 離開
p [some variable](print): 秀某個變數的值
n(next line): 下一行
c(continue): 繼續下去
s(step into): 進入函式
r(return): 到本函式的return敘述式
l(list): 秀出目前所在行號
!: 改變變數的值

Python print

- The simplest way to produce output is using the *print* statement where you can pass zero or more expressions separated by commas.
- This function converts the expressions you pass into a string and writes the result to standard output.



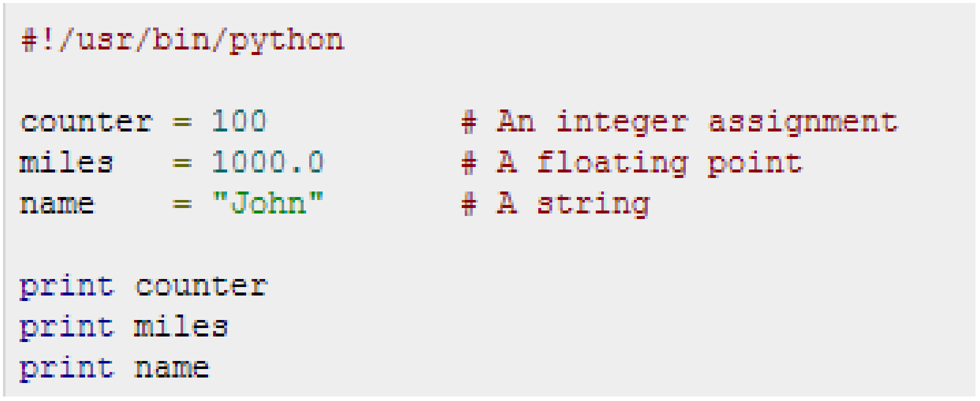
```
#!/usr/bin/python

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



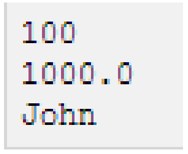
```
Python is really a great language, isn't it?
```

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.



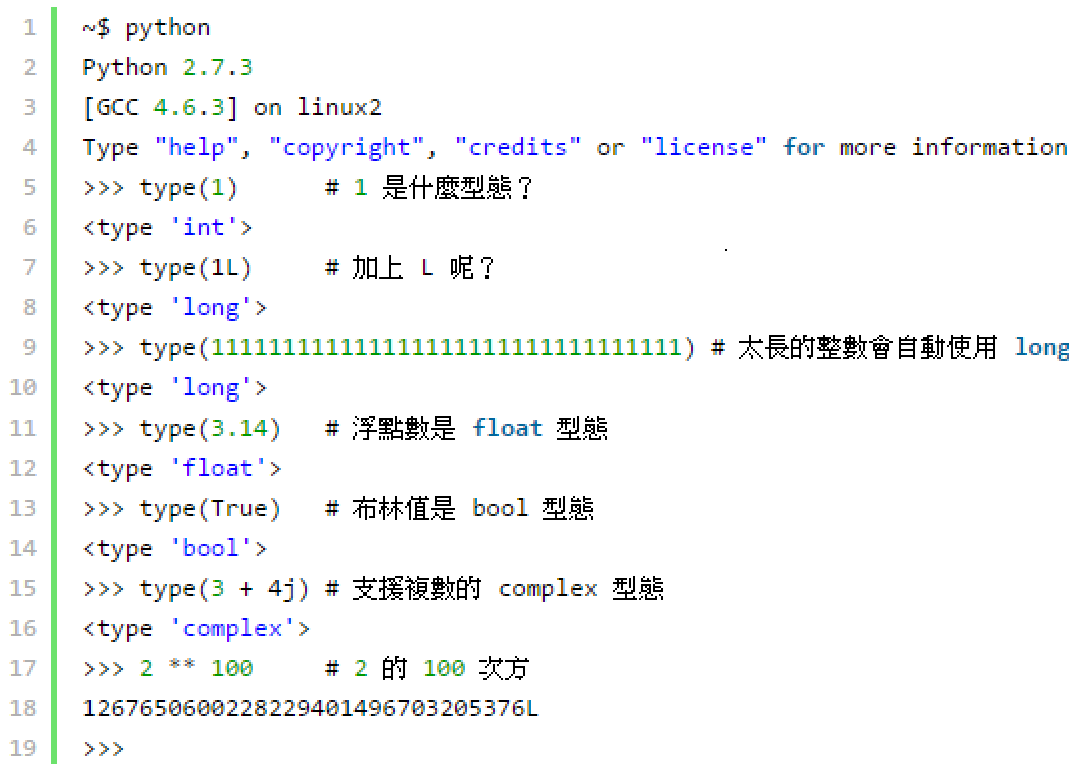
```
#!/usr/bin/python

counter = 100          # An integer assignment
miles   = 1000.0       # A floating point
name    = "John"       # A string

print counter
print miles
print name
```



```
100
1000.0
John
```

Python types

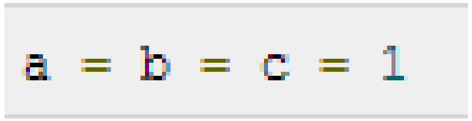
- Numeric type
  - int : 42 may be transparently expanded to long through 438324932L
  - long :
  - float : 2.171892
  - complex : 4 + 3j
  - bool : True of False



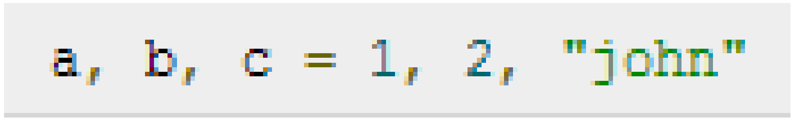
```
~$ python
Python 2.7.3
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information
>>> type(1)       # 1 是什麼型態？
<type 'int'>
>>> type(1L)      # 加上 L 呢？
<type 'long'>
>>> type(11111111111111111111111111111111) # 太長的整數會自動使用 long
<type 'long'>
>>> type(3.14)    # 浮點數是 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
>>>
```

Multiple Assignment

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

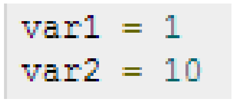


```
a = b = c = 1
```



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

Standard Data Types

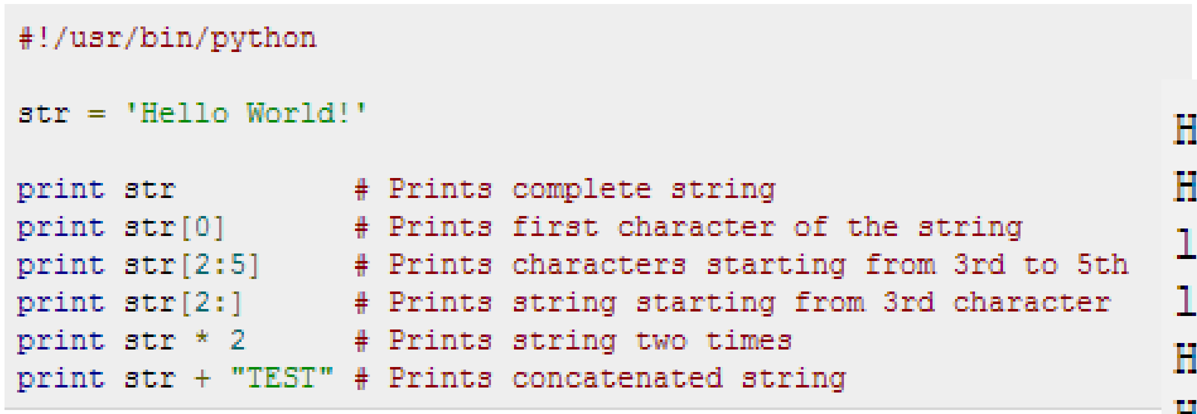
- Python has five standard data types:
  1. Numbers (Number data types store numeric values.) 
     ```
     var1 = 1
     var2 = 10
     ```
  2. String (Strings in Python are identified as a contiguous set of characters in between quotation marks(“ ”).)
  3. List (Lists are the most versatile of Python's 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.)

Python types

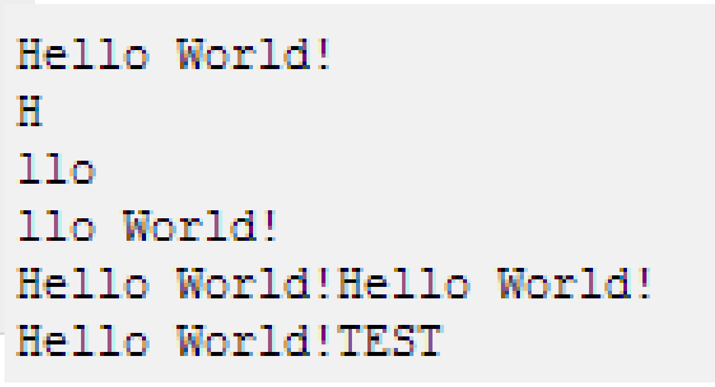
- Str – "Hello"
- List – [69, 6.9, 'mystring', True]
- Tuple – (69, 6.9, 'mystring', True) immutable
- Set/frozenset
  - set([69, 6.9, 'str', True])
  - frozenset([69, 6.9, 'str', True]) immutable –no duplicates & unordered
- Dictionary or hash – {'key 1': 6.9, 'key2': False} - group of key and value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

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.



```python
#!/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
```

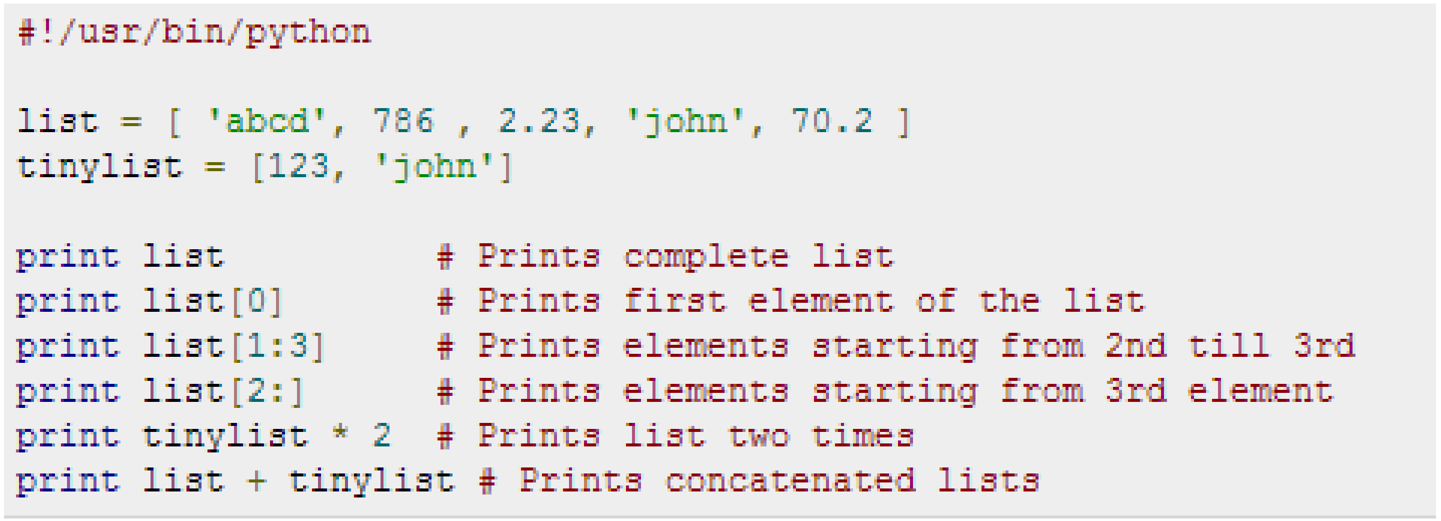


```
Hello World!
H
llo
llo World!
Hello World!Hello World!
Hello World!TEST
```

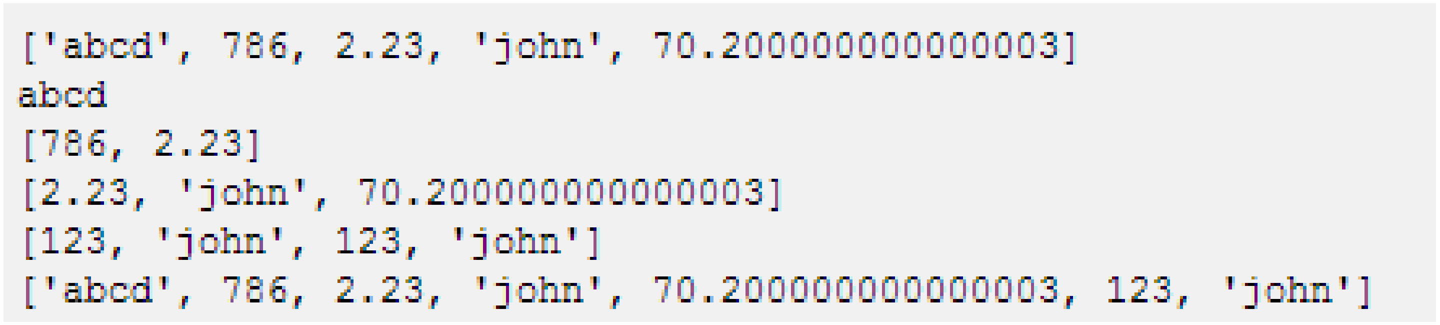
Python Lists

- 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 type.
- 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



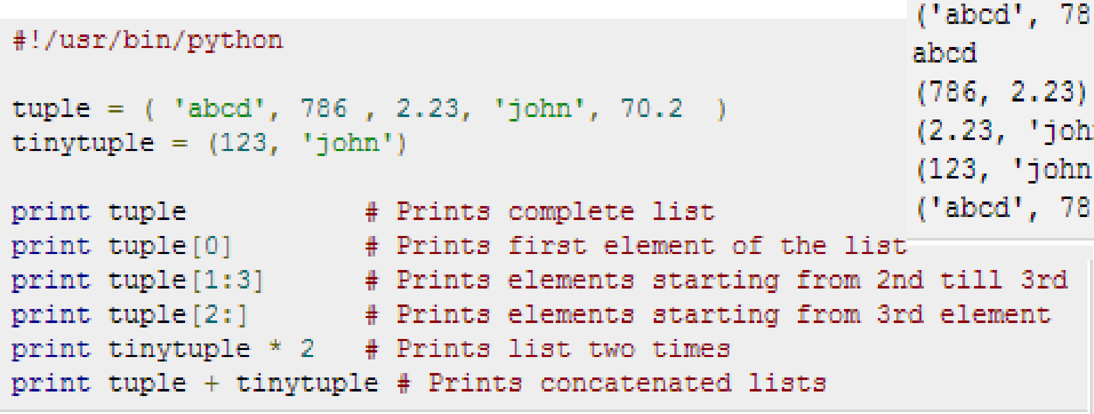
```
#!/usr/bin/python

list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
tinylist = [123, 'john']

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



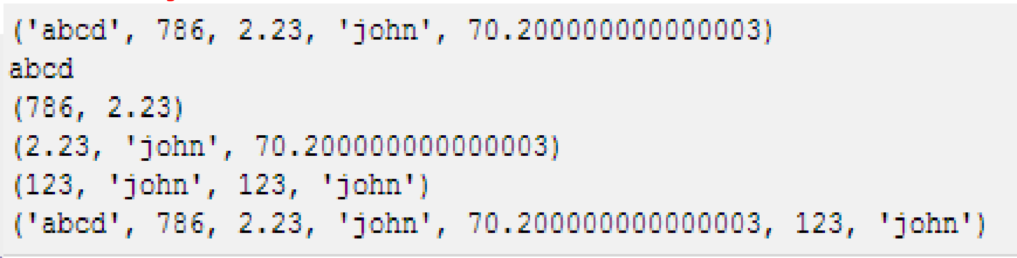
```
['abcd', 786, 2.23, 'john', 70.200000000000003]
abcd
[786, 2.23]
[2.23, 'john', 70.200000000000003]
[123, 'john', 123, 'john']
['abcd', 786, 2.23, 'john', 70.200000000000003, 123, 'john']
```

Python Tuples

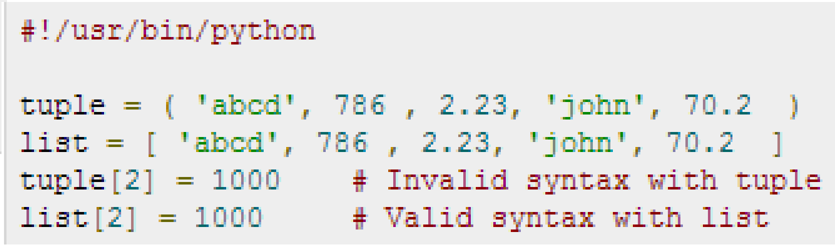
- A tuple consists of a number of values separated by commas .
- Tuples are enclosed within parentheses (**( )**).
- The main differences between lists and tuples are:
  - Lists are enclosed in brackets (**[ ]**) and their elements and size can be changed, while tuples are enclosed in parentheses (**( )**) 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
```

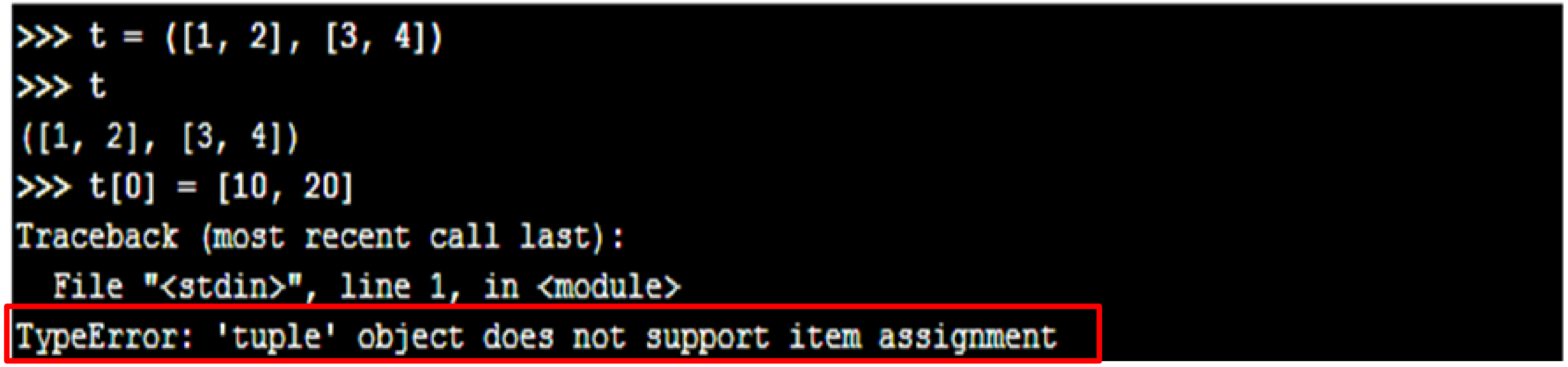


```
('abcd', 786, 2.23, 'john', 70.200000000000003)
abcd
(786, 2.23)
(2.23, 'john', 70.200000000000003)
(123, 'john', 123, 'john')
('abcd', 786, 2.23, 'john', 70.200000000000003, 123, 'john')
```



```
#!/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
```

Tuple Example

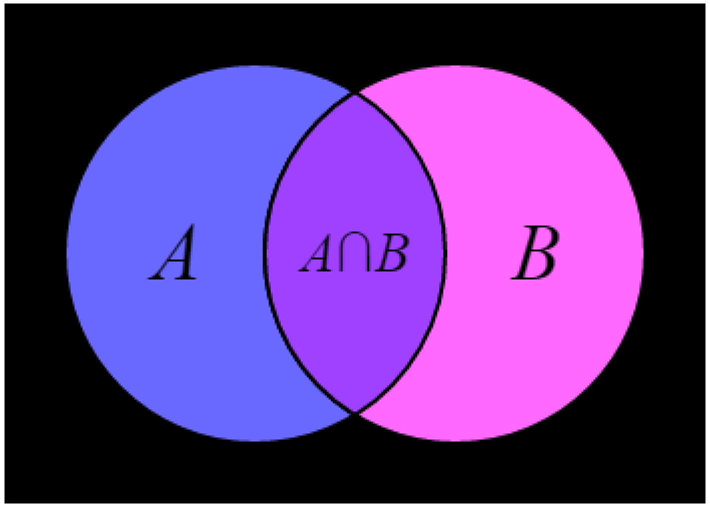


```
>>> t = ([1, 2], [3, 4])
>>> t
([1, 2], [3, 4])
>>> t[0] = [10, 20]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

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 me unique

- Checking membership
- Removing duplicates

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

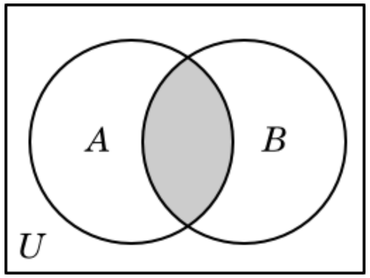
- in place

```
>>> 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])
```

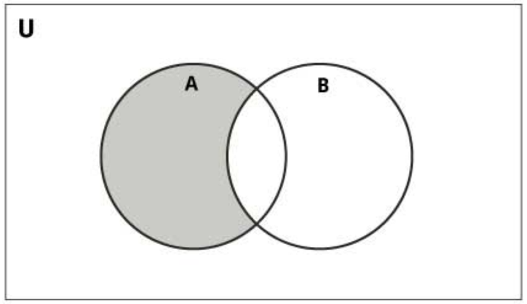
Mathematical operations

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

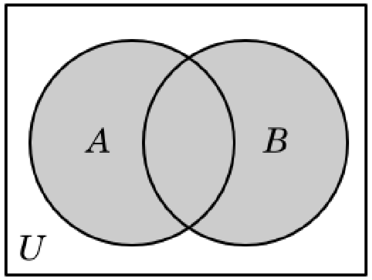
```
>>> a.intersection(b)
set([2, 3])
>>> a & b
set([2, 3])
```



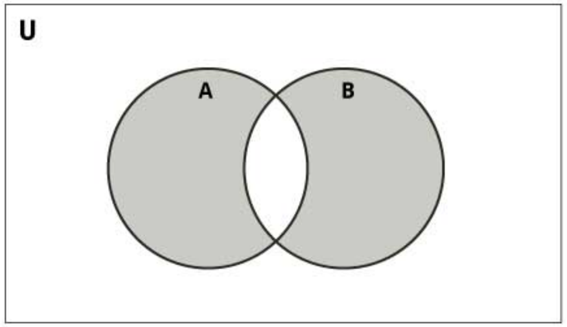
```
>>> a.difference(b)
set([1])
>>> a - b
set([1])
```



```
>>> 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])
```



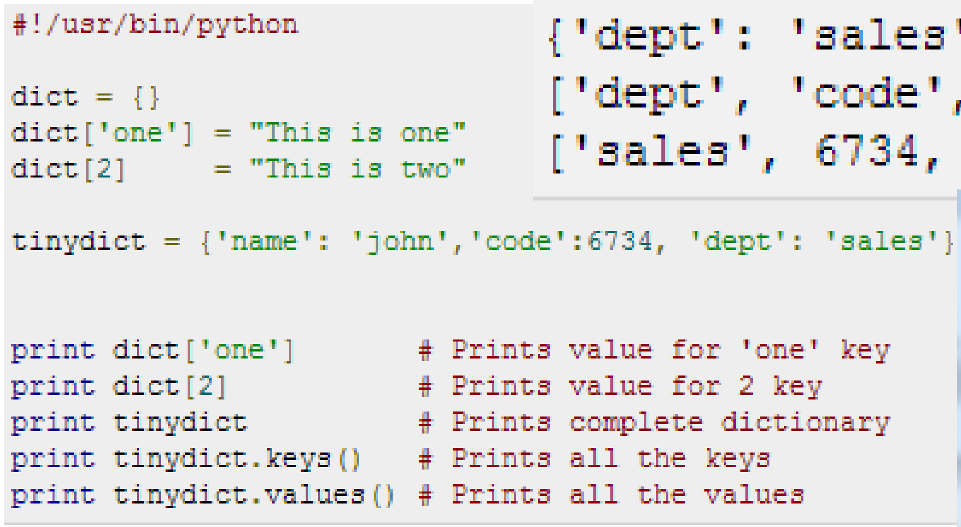
frozenset

- 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

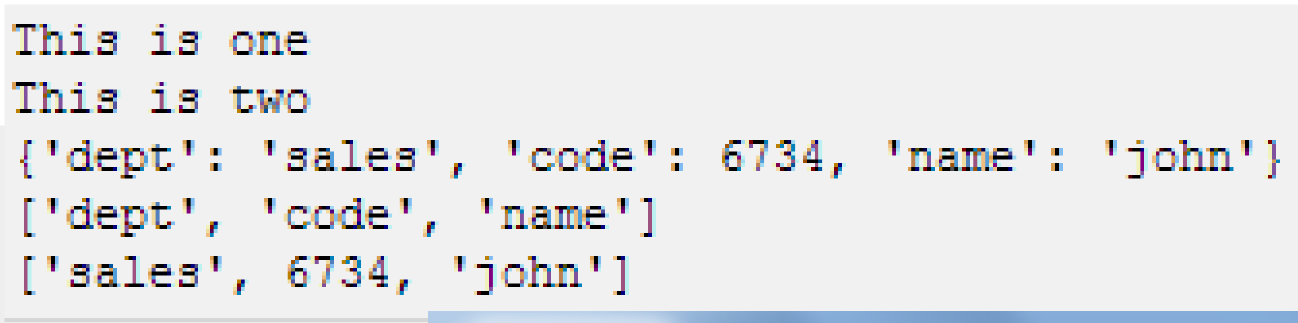
```
>>> a = set([1, 2, 3])
>>> b = set([2, 3, 4])

>>> 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])])
```

Python Dictionary

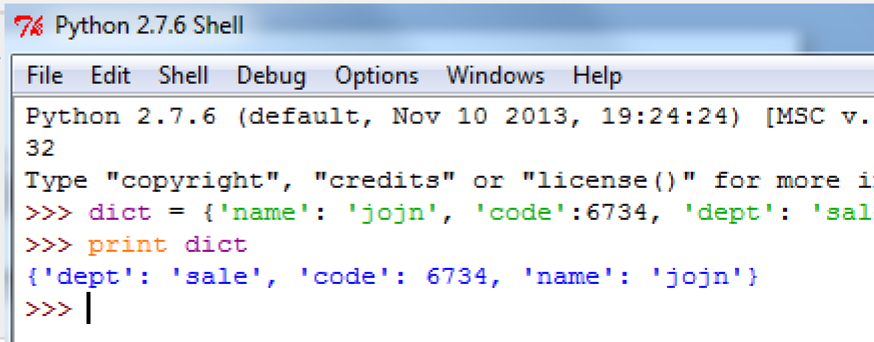
- A dictionary key can be almost any Python type, but are usually numbers or strings.
  - Values, on the other hand, can be any arbitrary Python object.
- Dictionaries are enclosed by curly braces ({ }) and values can be assigned and accessed using square braces ([])



```
#!/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
```



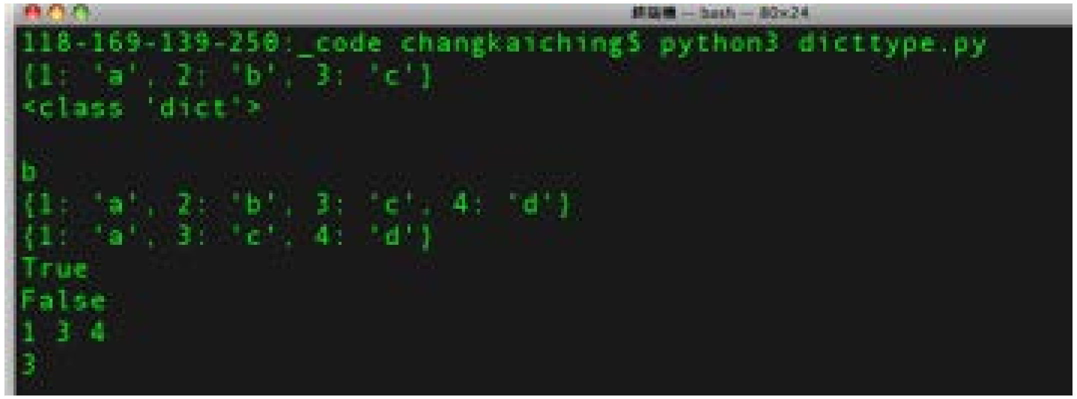
```
This is one
This is two
{'dept': 'sales', 'code': 6734, 'name': 'john'}
['dept', 'code', 'name']
['sales', 6734, 'john']
```



```
Python 2.7.6 (default, Nov 10 2013, 19:24:24) [MSC v.
32
Type "copyright", "credits" or "license()" for more i
>>> dict = {'name': 'jojn', 'code':6734, 'dept': 'sal
>>> print dict
{'dept': 'sale', 'code': 6734, 'name': 'jojn'}
>>> |
```

Dictionary – Python 3.x

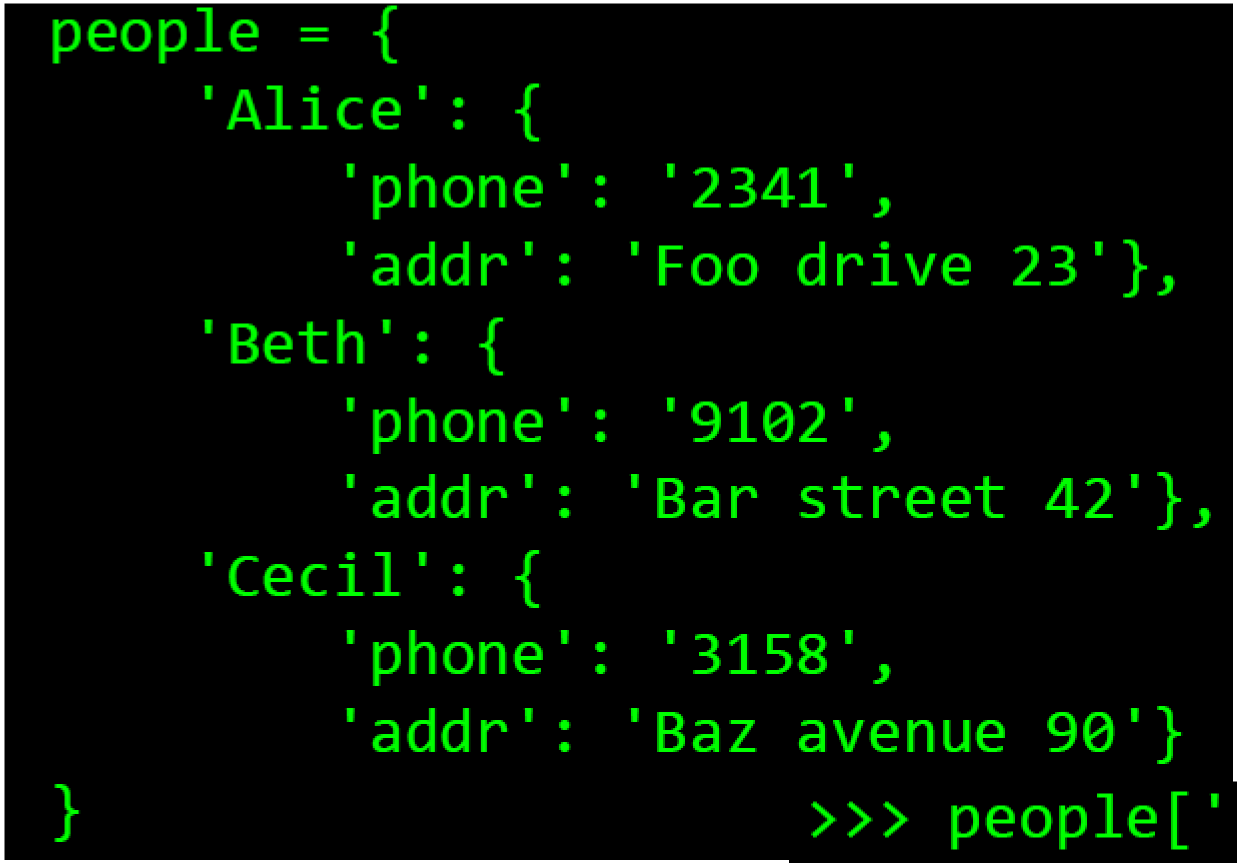
```python
d = {1:"a", 2:"b", 3:"c"}
print(d)
print(type(d))
print()

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()
```



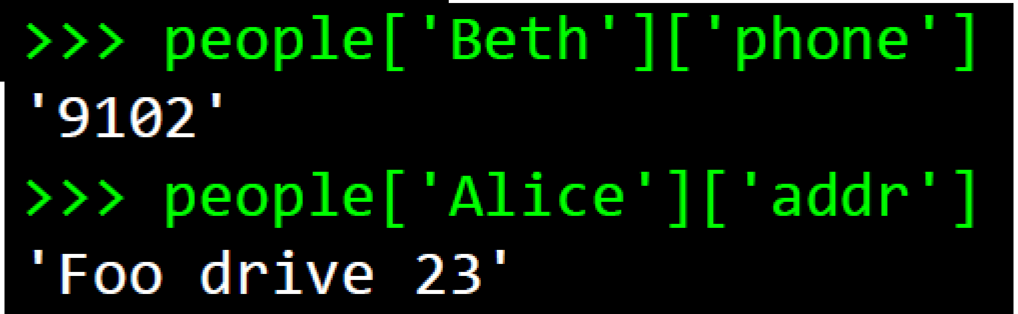
```
118-169-139-250:_code changkaiching$ python3 dicttype.py
{1: 'a', 2: 'b', 3: 'c'}
<class 'dict'>

b
{1: 'a', 2: 'b', 3: 'c', 4: 'd'}
{1: 'a', 3: 'c', 4: 'd'}
True
False
1 3 4
3
```

Nest Dictionary

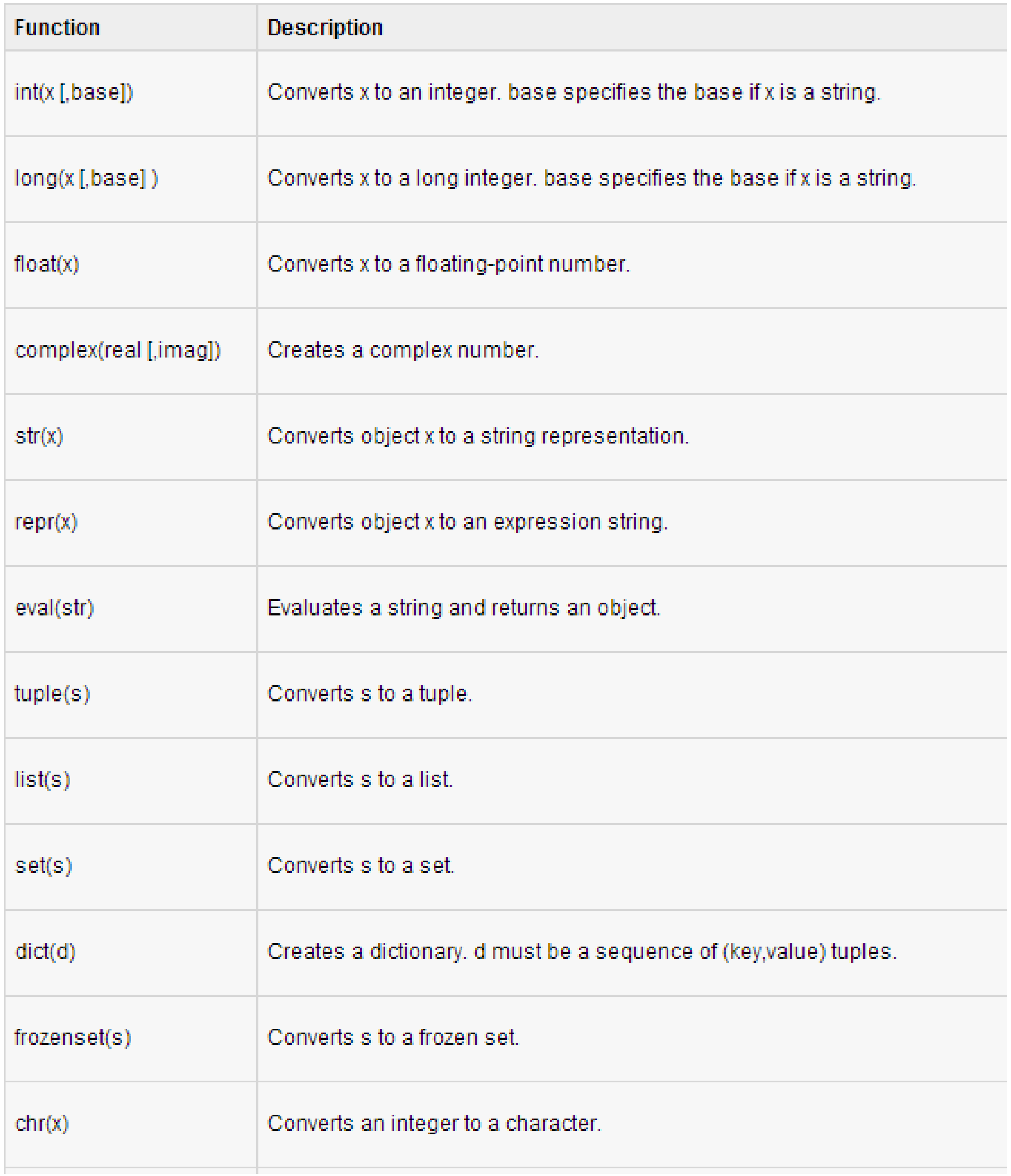
- Dictionary can be used as a tiny database.



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



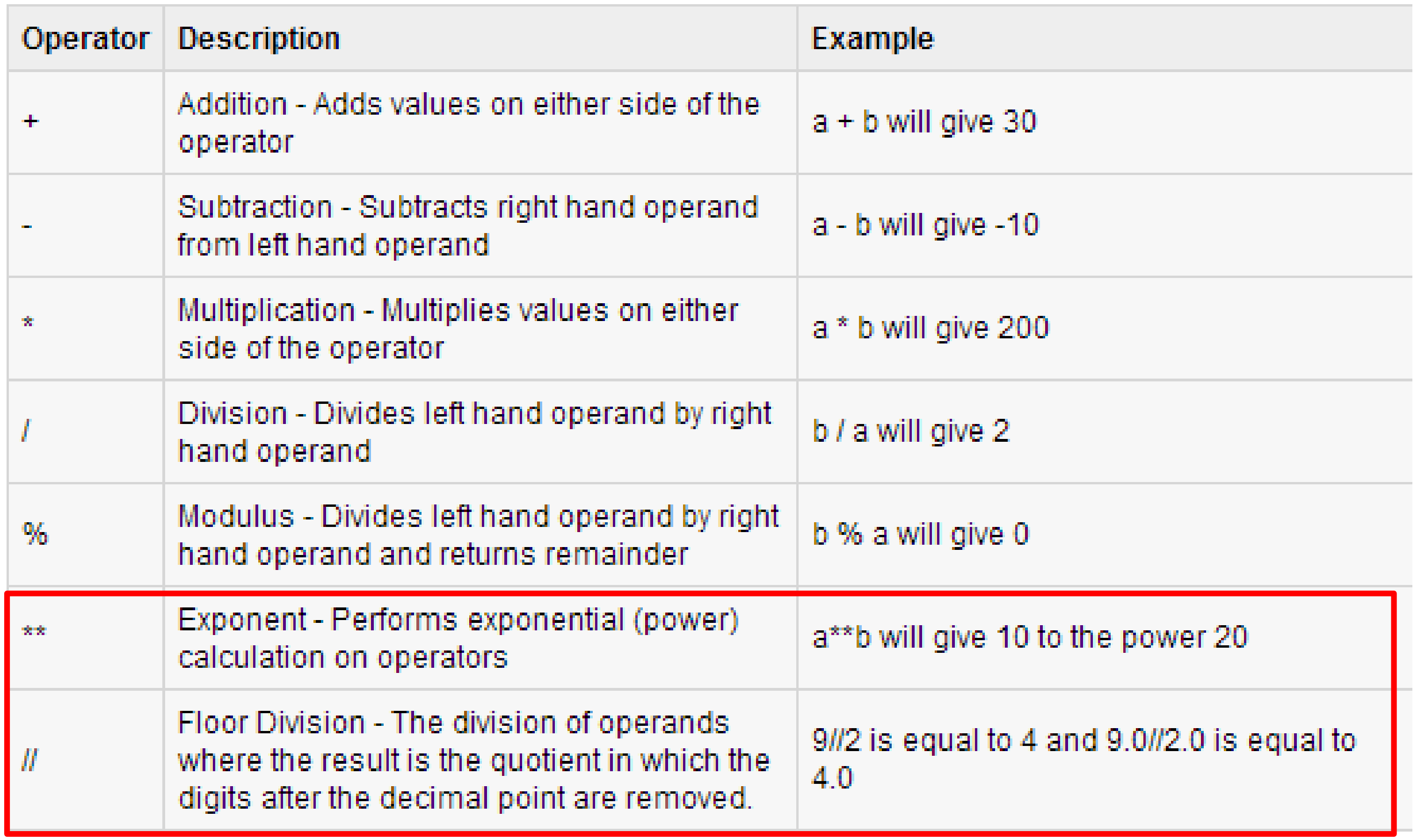
```
>>> people['Beth']['phone']
'9102'
>>> people['Alice']['addr']
'Foo drive 23'
```



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

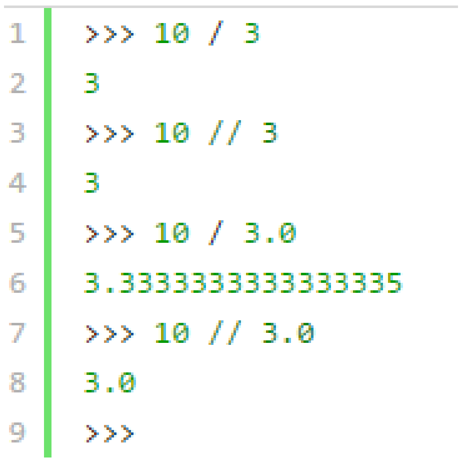
- Assume variable a holds 10 and variable b holds 20



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

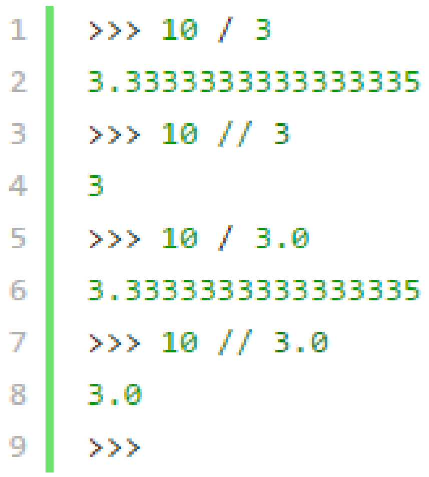
Example

- Python 2.7



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

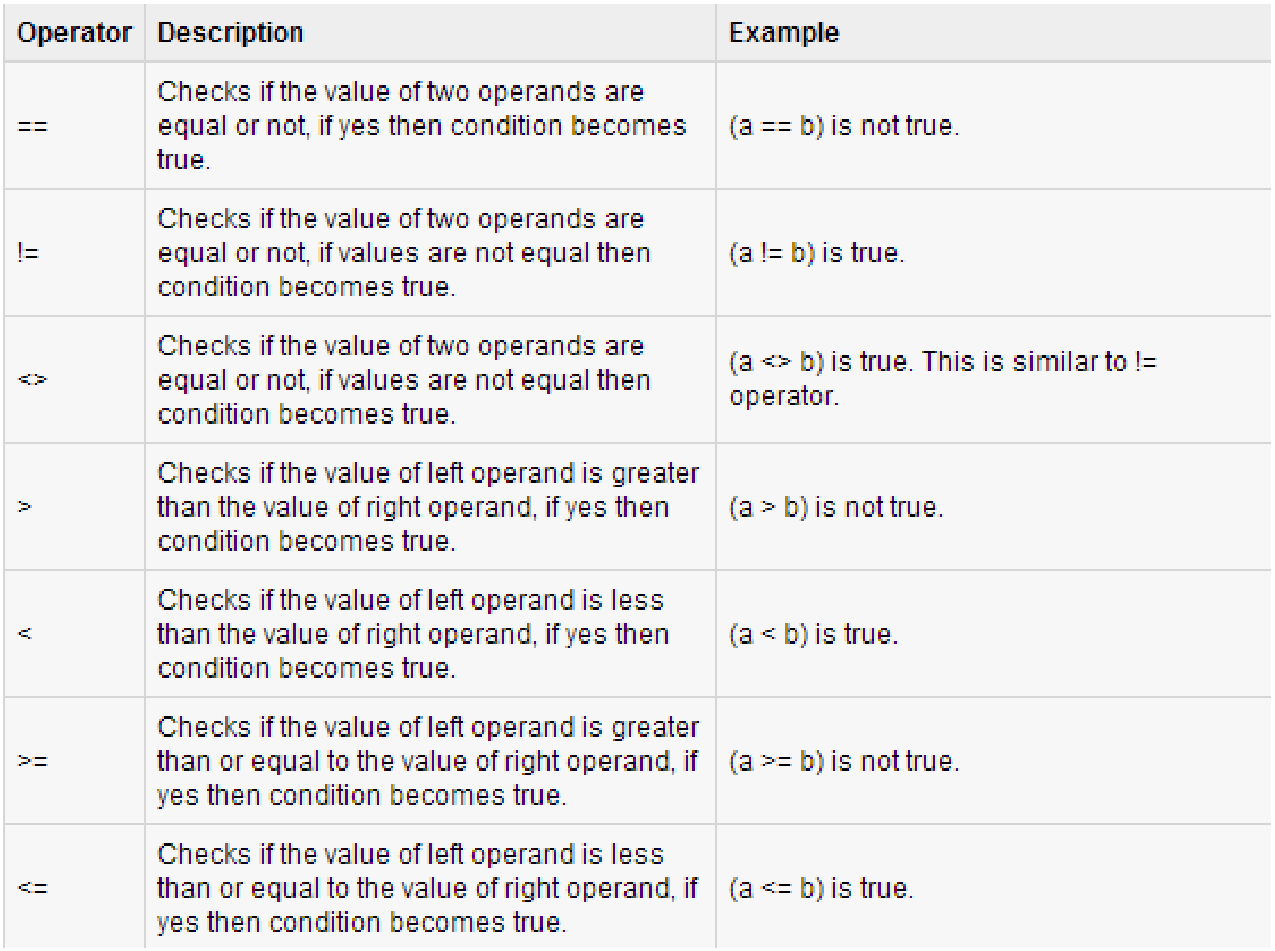
Python 3.3



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

Python Comparison Operators

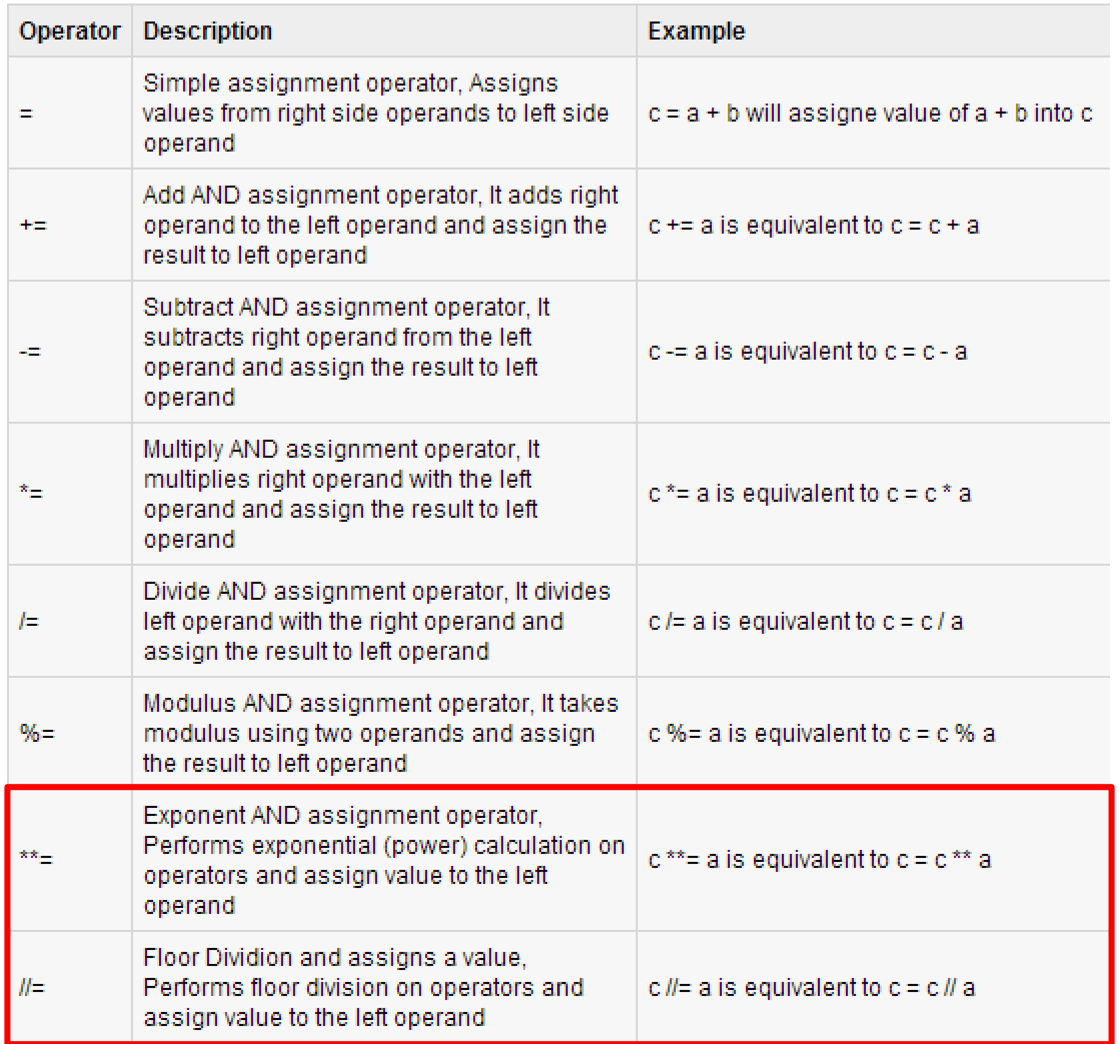
- Assume variable a holds 10 and variable b holds 20



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

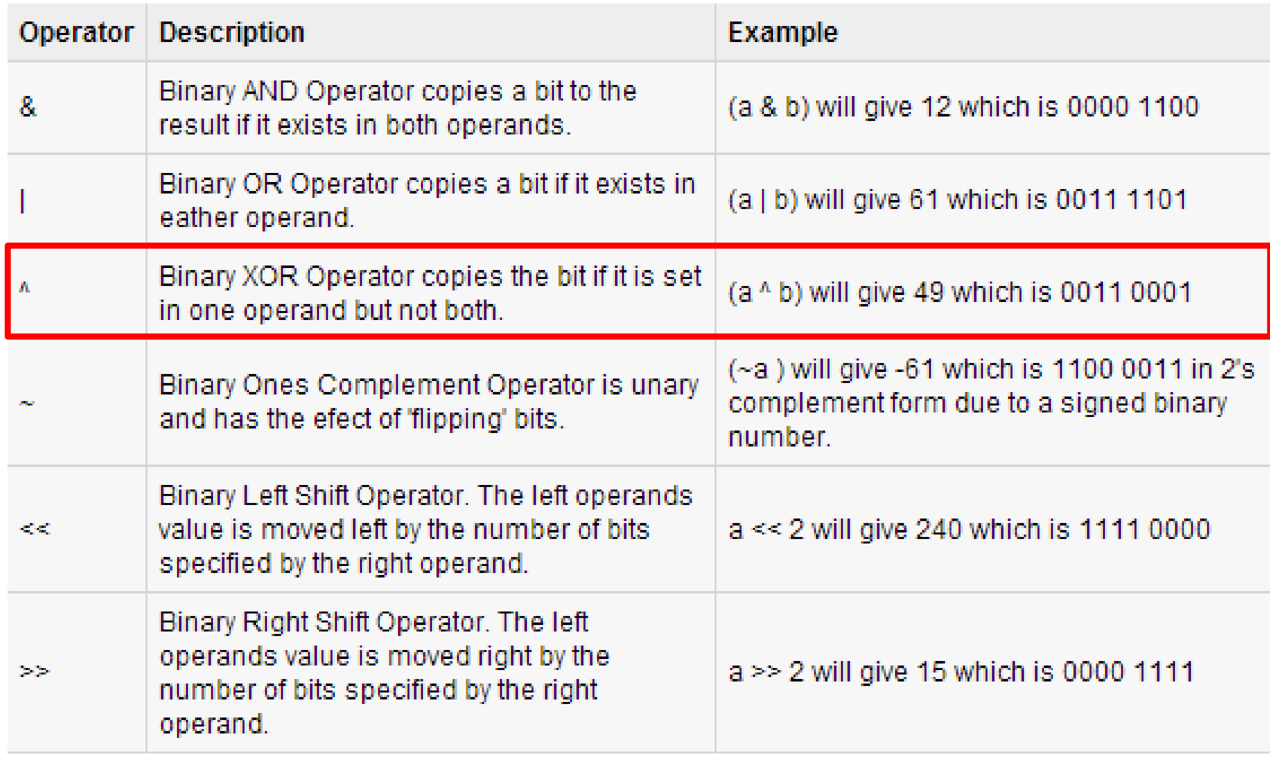
- Assume variable a holds 10 and variable b holds 20



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

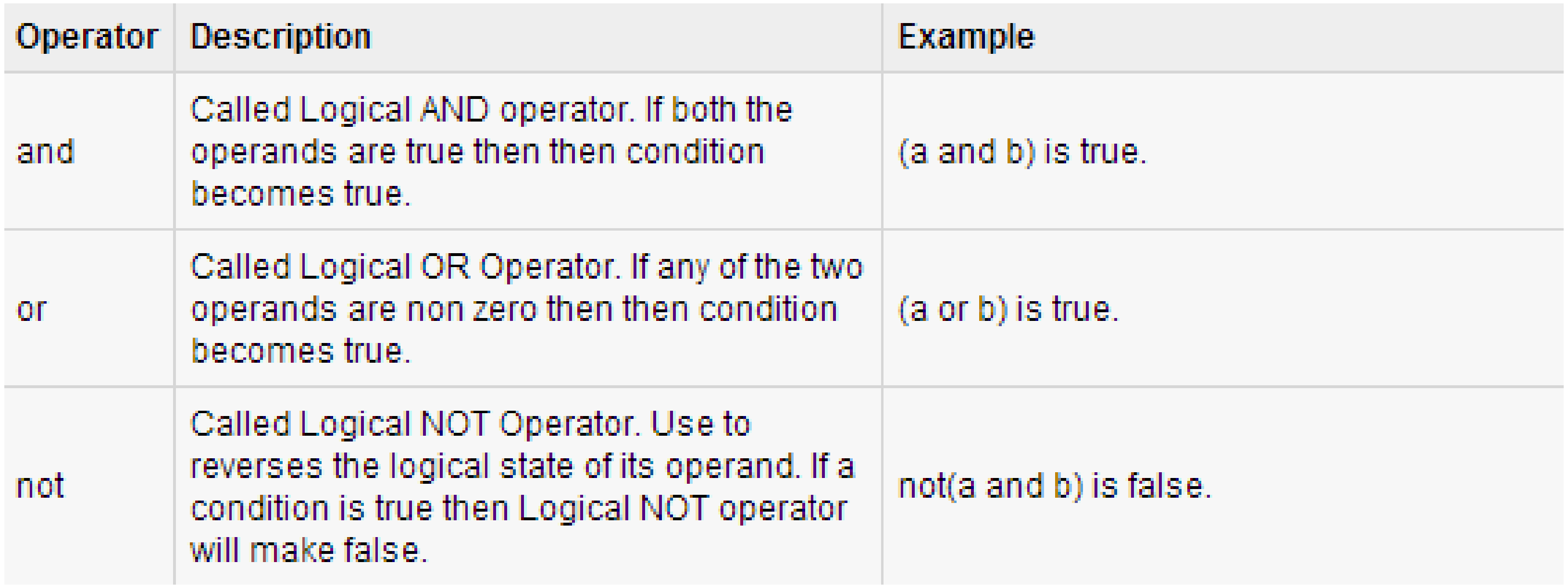
- Assume if a = 60; and b = 13;
- Now in binary format they will be as follows:
- a = 0011 1100;
- b = 0000 1101
- a&b = 0000 1100
- a|b = 0011 1101
- a^b = 0011 0001
- ~a = 1100 0011



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

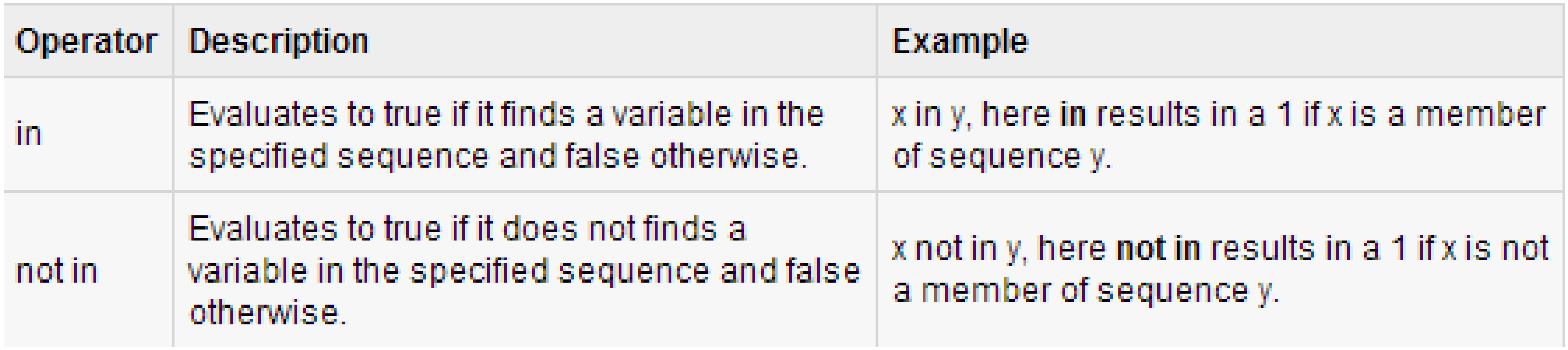
- Assume variable a holds 10 and variable b holds 20



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



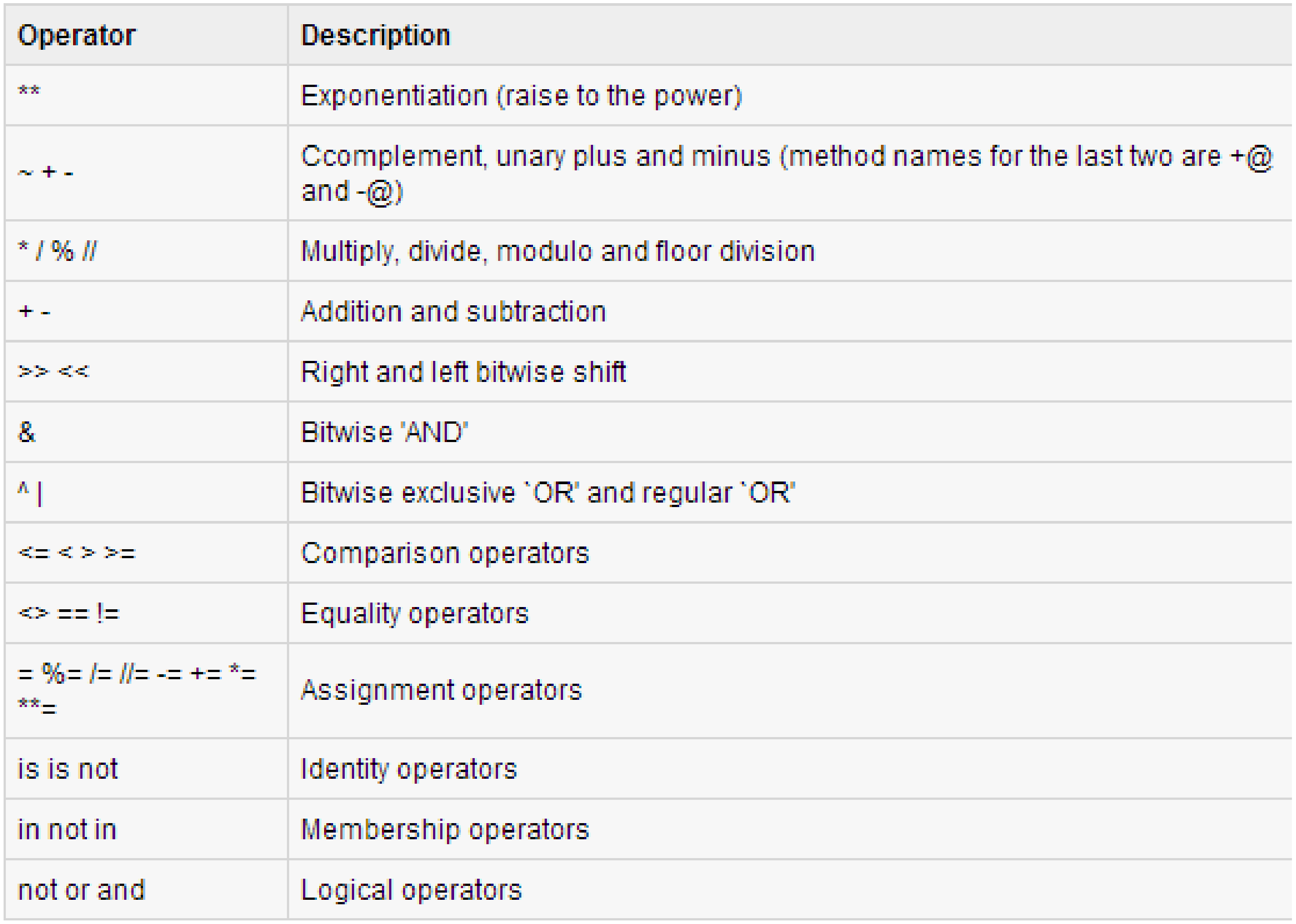
Operator	Description	Example
in	Evaluates to true if it finds a variable in the specified sequence and false otherwise.	x in y, here **in** results in a 1 if x is a member of sequence y.
not in	Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.	x not in y,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



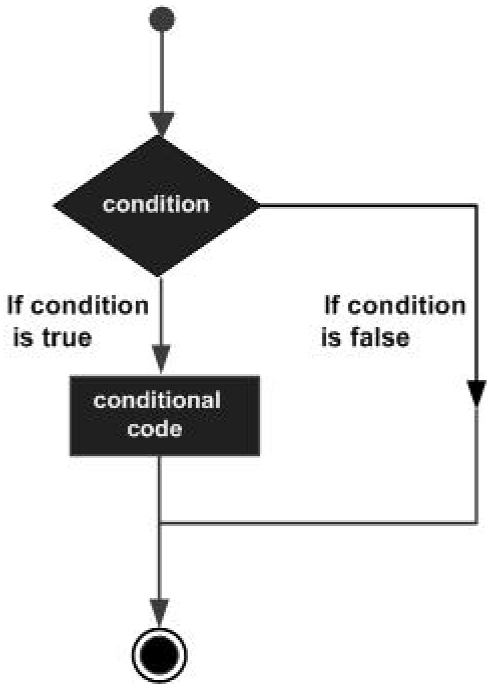
Operator	Description
**	Exponentiation (raise to the power)
~ + -	Ccomplement, unary plus and minus (method names for the last two are +@ and -@)
* / % //	Multiply, divide, modulo and floor division
+ -	Addition and subtraction
>> <<	Right and left bitwise shift
&	Bitwise 'AND'
^ |	Bitwise exclusive `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”
  ```

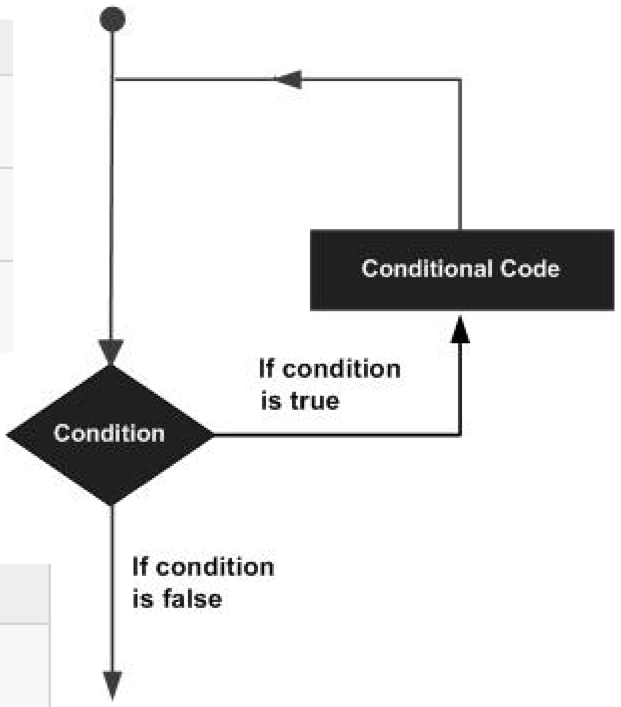
- **if** (job == "millionaire") **or** (state != "dead"):

  ```
          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)
  ```

-
  ```
  for i in range(10):
          print i
  for i in xrange(1000): # does not allocate all initially
          print i
  ```

-
  ```
  while True:
          pass
  ```

-
  ```
  for i in xrange(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
  ...

range() and xrange()

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

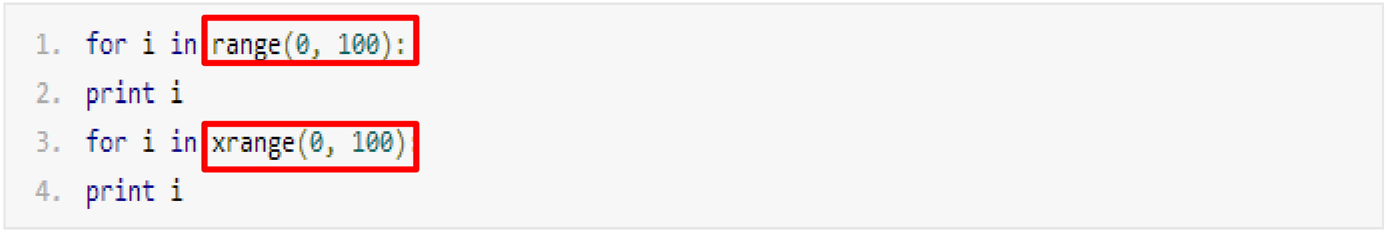
```
>>> range(5)
[0, 1, 2, 3, 4]
>>> range(1,5)
[1, 2, 3, 4]
>>> range(0,6,2)
[0, 2, 4]
```

- xrange() return a generator

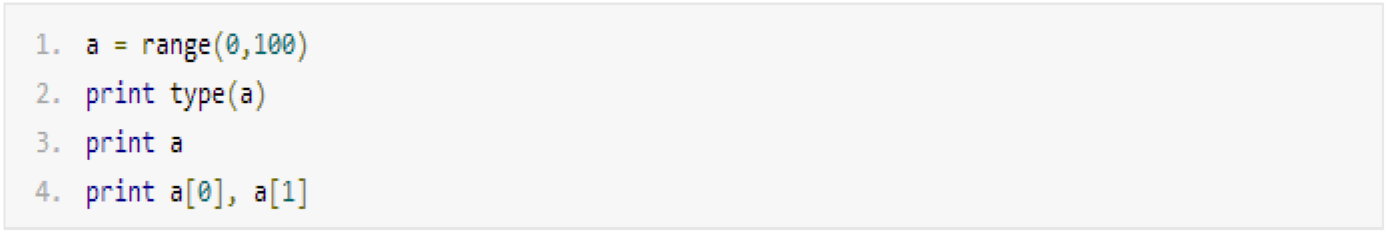
```
>>> xrange(5)
xrange(5)
>>> list(xrange(5))
[0, 1, 2, 3, 4]
>>> xrange(1,5)
xrange(1, 5)
>>> list(xrange(1,5))
[1, 2, 3, 4]
>>> xrange(0,6,2)
xrange(0, 6, 2)
>>> list(xrange(0,6,2))
[0, 2, 4]
```

Difference between range() and xrange()

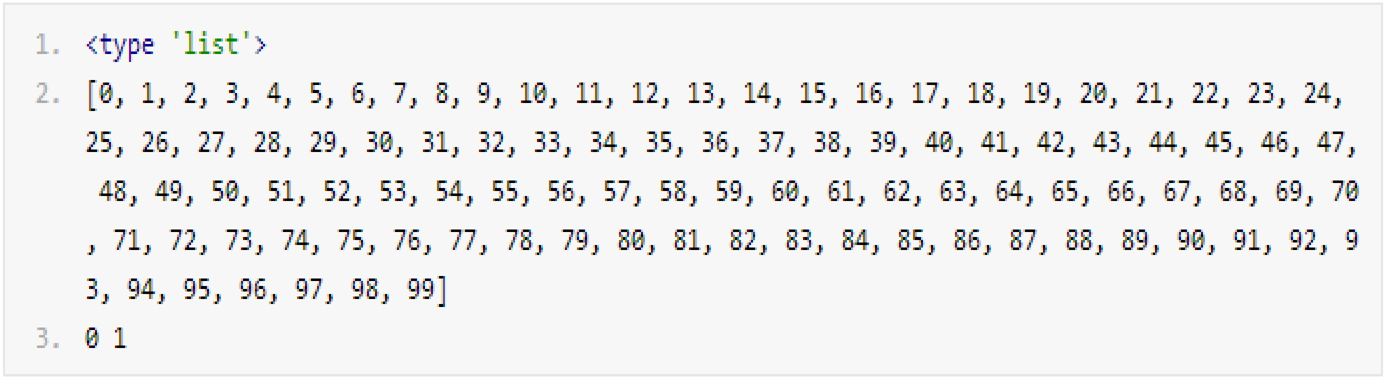
- range()



```
for i in range(0, 100):
print i
for i in xrange(0, 100)
print i
```



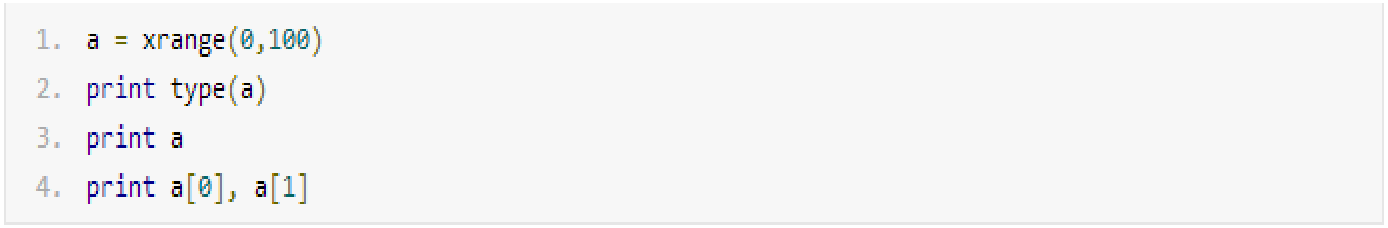
```
a = range(0,100)
print type(a)
print a
print a[0], a[1]
```



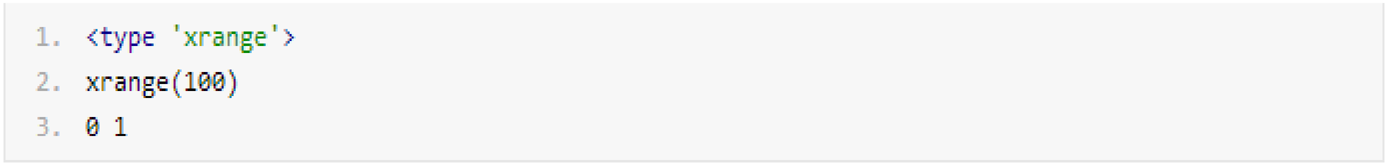
```
<type 'list'>
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24,
25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47,
 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70
, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 9
3, 94, 95, 96, 97, 98, 99]
0 1
```

Difference of range() and xrange()

- xrange()



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



```
<type 'xrange'>
xrange(100)
0 1
```

Python Exceptions Handling

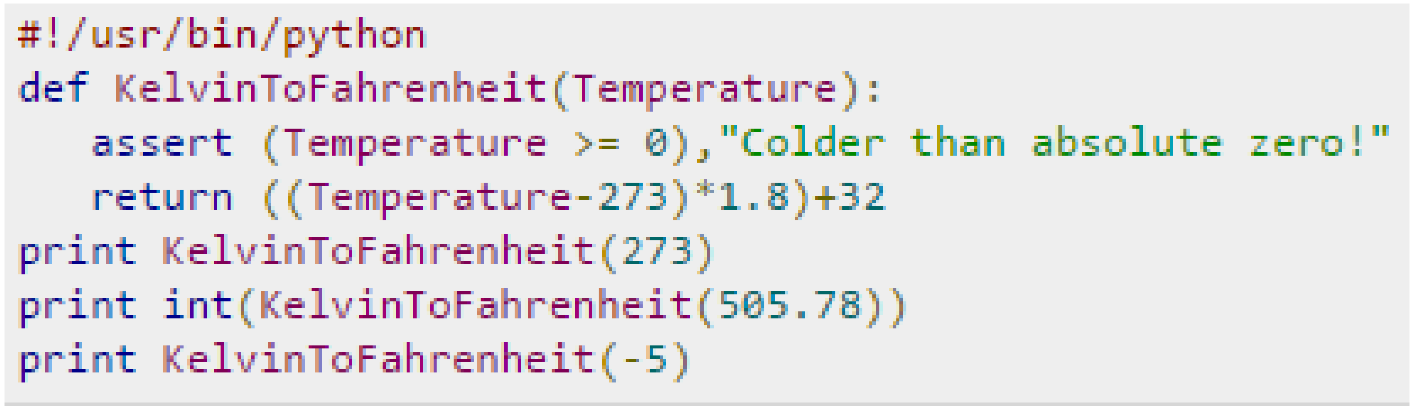
- Python provides two very important features to handle any unexpected error and to add debugging capabilities in them.
  - **Exception Handling**
  - **Assertions**

Assertions in Python

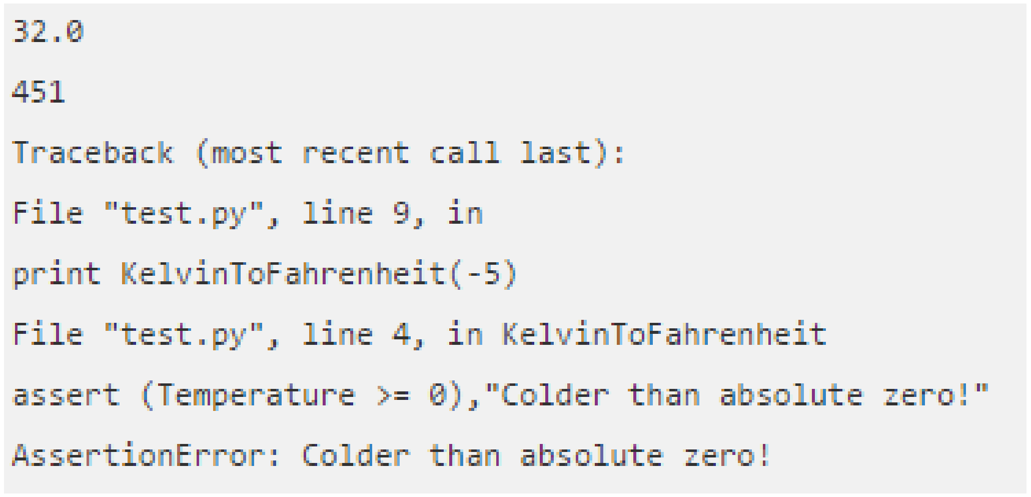
- An assertion is a sanity-check that you can turn on or turn off when you are done with your testing of the program.
- The easiest way to think of an assertion is to liken it to a **raise-if** statement (or to be more accurate, a raise-if-not statement).
- An expression is tested, and if the result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

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

Example



```
#!/usr/bin/python
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)
```



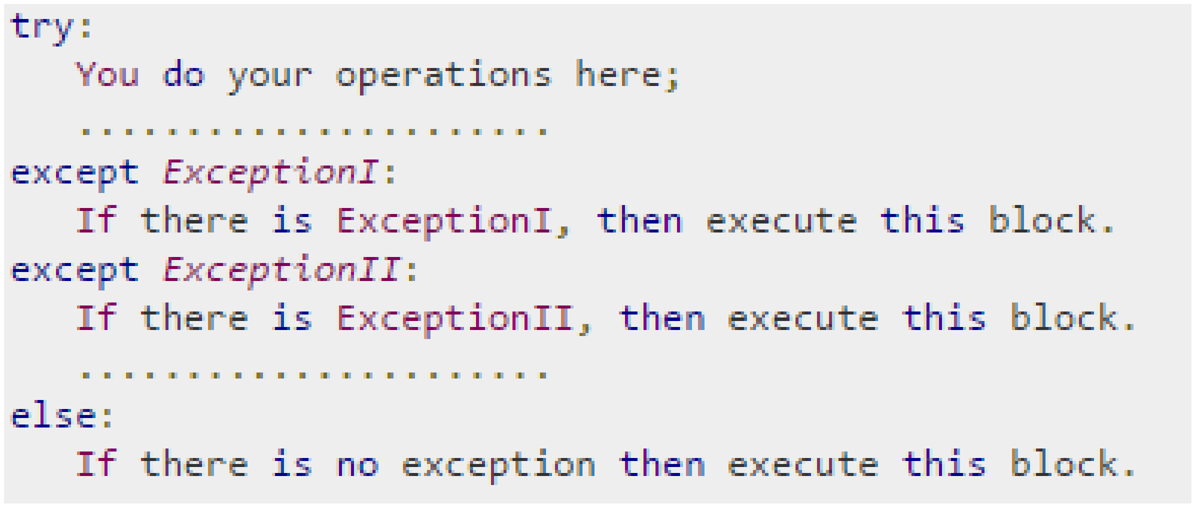
```
32.0
451
Traceback (most recent call last):
File "test.py", line 9, in
print KelvinToFahrenheit(-5)
File "test.py", line 4, in KelvinToFahrenheit
assert (Temperature >= 0),"Colder than absolute zero!"
AssertionError: Colder than absolute zero!
```

What is Exception?

- An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions.
- In general, when a Python script encounters a situation that it cannot cope with, 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

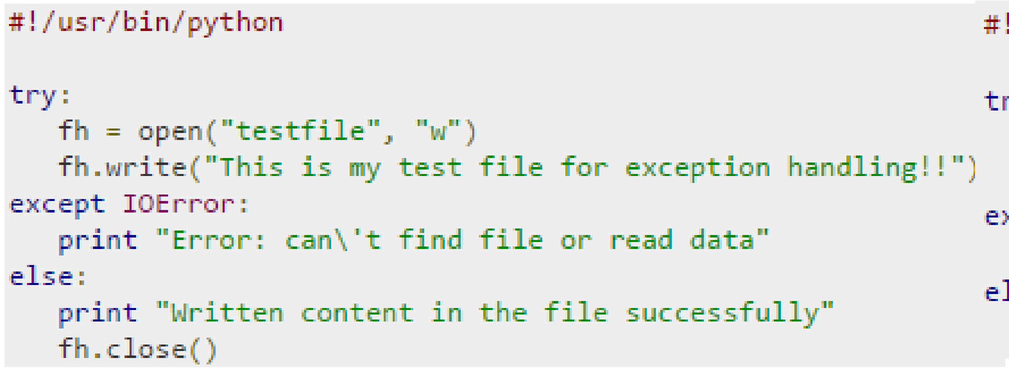
- Here are few important points about the above-mentioned syntax
  - A single try statement can have multiple except statements.
    - This is useful when the try block contains statements that may throw different types of exceptions.
  - You can also provide a generic except clause, which handles any 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.

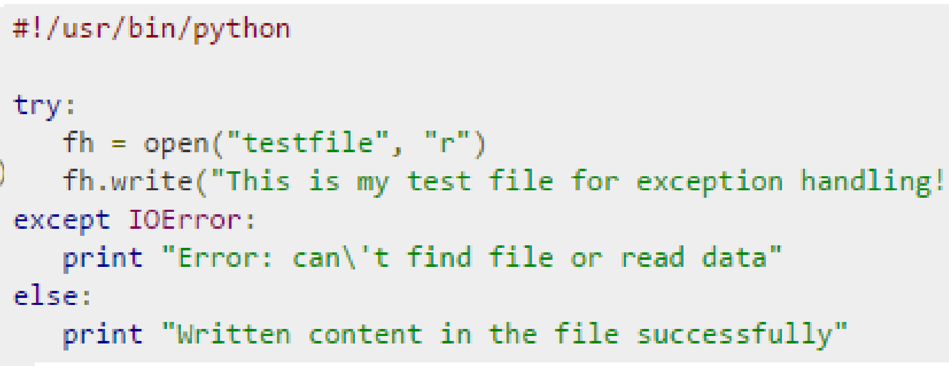
Example

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

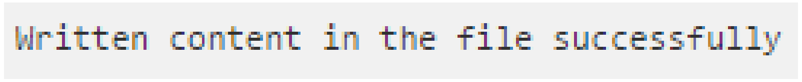
This example tries to open a file where you do not have write permission, so it raises an exception



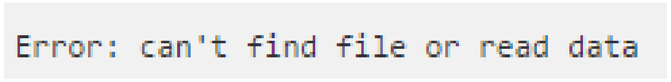
```
#!/usr/bin/python

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()
```



```
#!/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"
```

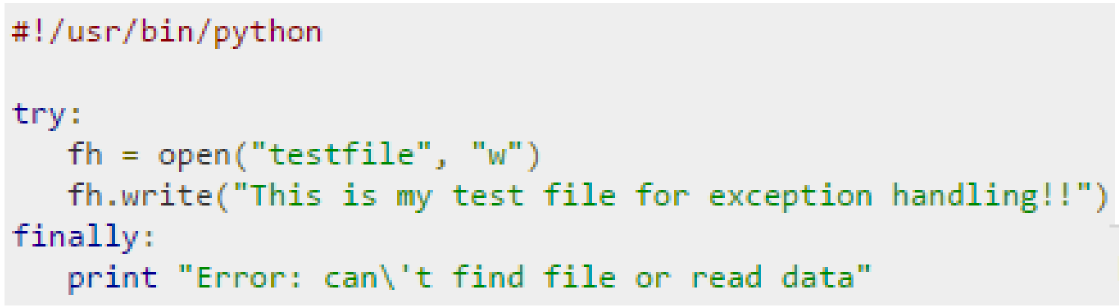


```
Written content in the file successfully
```

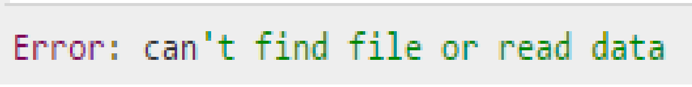


```
Error: can't find file or read data
```

The try-finally Clause

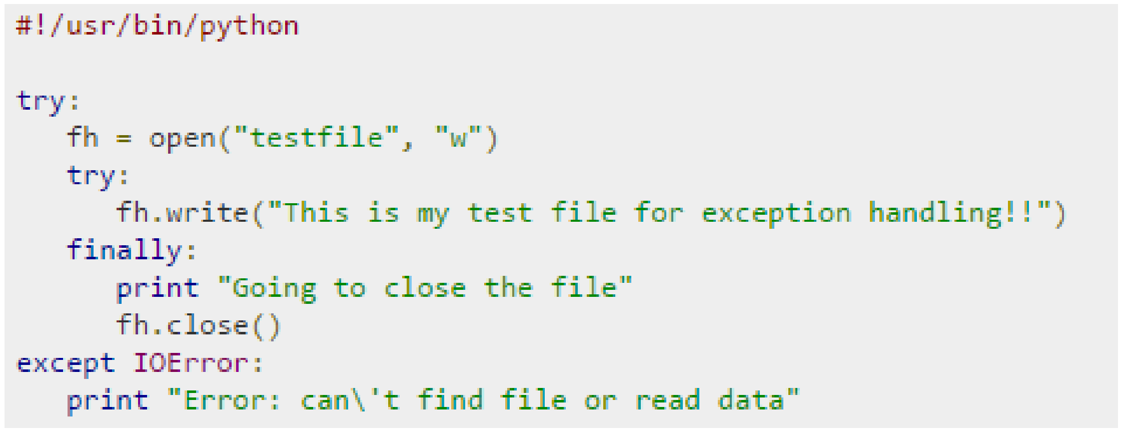


```
#!/usr/bin/python

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
```

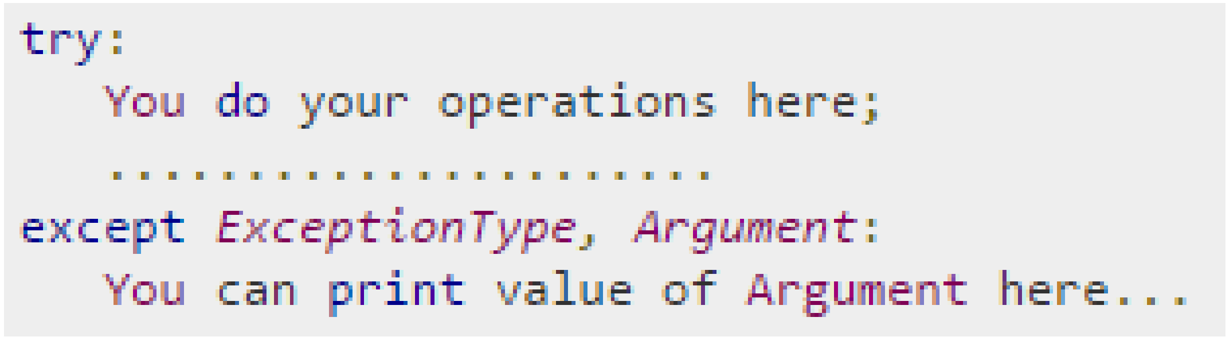
- Same example can be written more cleanly as follows



```
#!/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"
```

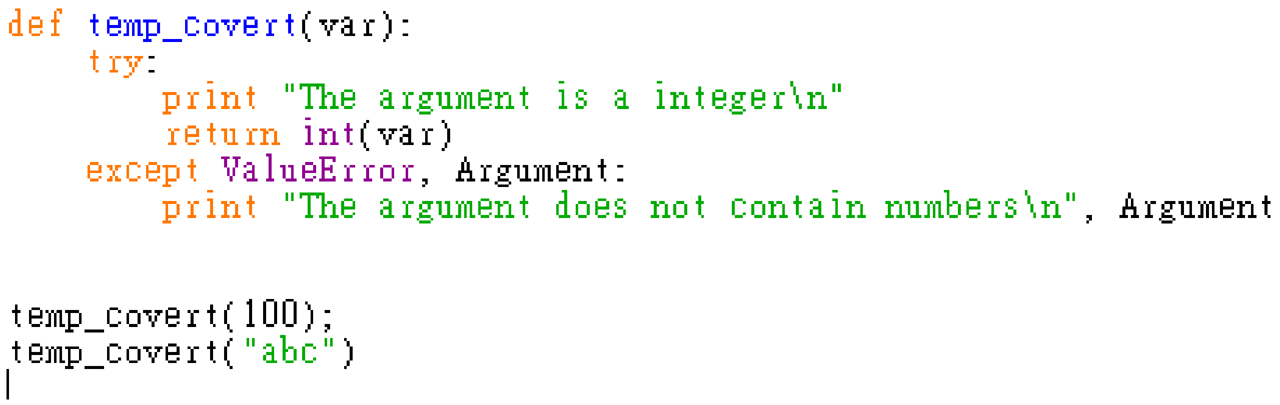
Argument of an Exception

- 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

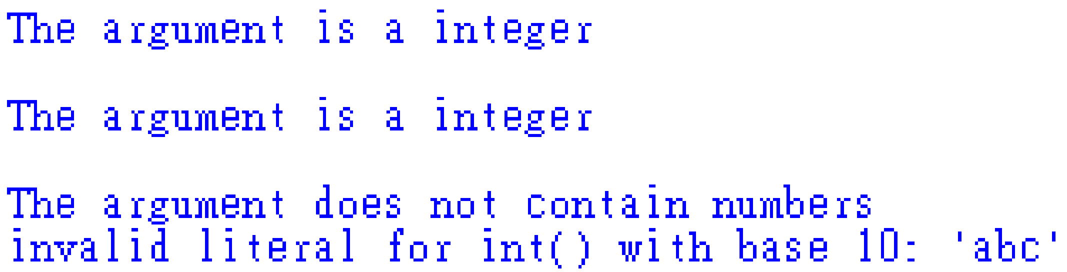


```
try:
   You do your operations here;
   ......................
except ExceptionType, Argument:
   You can print value of Argument here...
```

Example



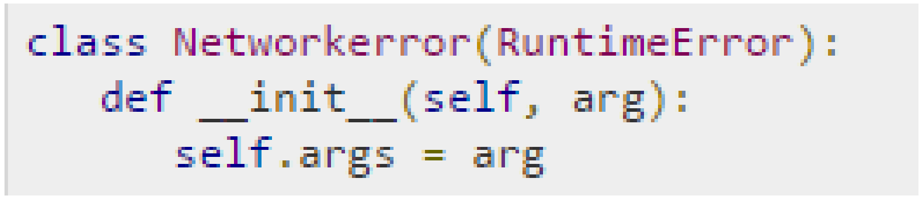
```python
def temp_covert(var):
    try:
        print "The argument is a integer\n"
        return int(var)
    except ValueError, Argument:
        print "The argument does not contain numbers\n", Argument


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



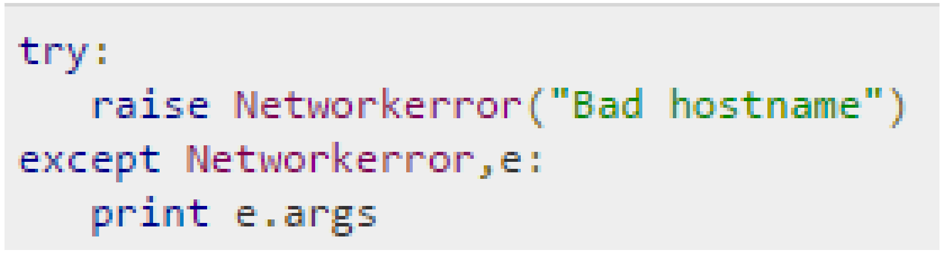
```
The argument is a integer

The argument is a integer

The argument does not contain numbers
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*. This is useful when you need to display more specific information when an exception is caught.



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



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