6. Data Types and Dynamic Typing (Cont.)

6.5 Strings

Strings can be delimited by a pair of single quotes (`'...'`), double quotes (`"..."`), triple single quotes (`'''...'''`), or triple double quotes (`"""..."""`). In Python, single-quoted string is the SAME as double-quoted string.
To place a single quote (`'`) inside a single-quoted string, you need to use escape sequence `\'`. Similarly, to place a double quote (`"`) inside a double-quoted string, use `\"`. There is no need for escape sequence to place a single quote inside a double-quoted string; or a double quote inside a single-quoted string.
A triple-single-quoted or triple-double-quoted string can *span multiple lines*. There is no need for escape sequence to place a single/double quote inside a triple-quoted string. Triple-quoted strings are useful for multi-line documentation, HTML and other codes.

Python 3 uses Unicode character set.

```
>>> s1 = 'apple'
>>> s1
'apple'
>>> s2 = "orange"
>>> s2
'orange'
>>> s3 = "'orange'"   # Escape sequence not required
>>> s3
"'orange'"
>>> s3 ="\"orange\""  # Escape sequence needed
>>> s3
'"orange"'

# A triple-single/double-quoted string can span multiple lines
>>> s4 = """testing
testing"""
>>> s4
'testing\ntesting'
```

Escape Sequences

Like C/C++/Java, you need to use escape sequences (a back-slash + a code):

- for special non-printable characters, such as tab (`\t`), newline (`\n`), carriage return (`\r`); and

Strings are Immutable

Strings are *immutable*, i.e., their contents cannot be modified. **String functions such as `upper()`, `replace()` returns a new string object instead of modifying the string under operation.**

Built-in Functions and Operators for Strings

You can operate on strings using:

- built-in functions such as `len()`;
- operators such as `in` (contains), `+` (concatenation), `*` (repetition), indexing `[i]`, and slicing `[m:n:step]`.

Note: These functions and operators are applicable to all `sequence` data types including `string`, `list`, and `tuple` (to be discussed later).

Function/Operator	Description	Examples `s = 'Hello'`
`len()`	Length	`len(s) ⇒ 5`

`in`	Contain?	`'ell' in s ⇒ True`
		`'he' in s ⇒ False`

`+`	Concatenation	`s + '!' ⇒ 'Hello!'`
`*`	Repetition	`s * 2 ⇒ 'HelloHello'`

`[i], [-i]`	Indexing to get a character.
	The front index begins at `0`; back index begins at -1 (=`len()-1`).
		`s[1] ⇒ 'e'`
		`s[-4] ⇒ 'e'`

`[m:n], [m:], [:n], [m:n:step]`	Slicing to get a substring.
	From index `m` (included) to `n` (excluded) with an optional `step` size.
	The default `m=0`, `n=-1`, `step=1`.
		`s[1:3] ⇒ 'el'`
		`s[1:-2] ⇒ 'el'`
		`s[3:] ⇒ 'lo'`
		`s[:-2] ⇒ 'Hel'`
		`s[:] ⇒ 'Hello'`
		`s[0:5:2] ⇒ 'Hlo'`

For examples,

```
>>> s = "Hello, world"   # Assign a string literal to the variable s
>>> type(s)              # Get data type of s
<class 'str'>
>>> len(s)       # Length
12
>>> 'ello' in s  # The in operator
True

# Indexing
>>> s[0]       # Get character at index 0; index begins at 0
'H'
```

```
>>> s[1]
'e'
>>> s[-1]      # Get Last character, same as s[len(s) - 1]
'd'
>>> s[-2]      # 2nd last character
'l'

# Slicing
>>> s[1:3]     # Substring from index 1 (included) to 3 (excluded)
'el'
>>> s[1:-1]
'ello, worl'
>>> s[:4]      # Same as s[0:4], from the beginning
'Hell'
>>> s[4:]      # Same as s[4:-1], till the end
'o, world'
>>> s[:]       # Entire string; same as s[0:len(s)]
'Hello, world'

# Concatenation (+) and Repetition (*)
>>> s = s + " again"  # Concatenate two strings
>>> s
'Hello, world again'
>>> s * 3             # Repeat 3 times
'Hello, world againHello, world againHello, world again'

# String is immutable
>>> s[0] = 'a'
TypeError: 'str' object does not support item assignment
```

Character Type?

Python does not have a dedicated character data type. A character is simply a string of length 1. You can use the indexing operator to extract individual character from a string, as shown in the above example; or process individual character using `for-in` loop (to be discussed later).

The built-in functions `ord()` and `chr()` operate on character, e.g.,

```
# ord(c) returns the integer ordinal (Unicode) of a one-character string
>>> ord('A')
65
>>> ord('水')
27700

# chr(i) returns a one-character string with Unicode ordinal i; 0 <= i <= 0x10ffff.
>>> chr(65)
'A'
>>> chr(27700)
'水'
```

Unicode vs ASCII

In Python 3, strings are defaulted to be Unicode. ASCII strings are represented as byte strings, prefixed with `b`, e.g., `b'ABC'`.

In Python 2, strings are defaulted to be ASCII strings (byte strings). Unicode strings are prefixed with `u`.

String-Specific Member Functions

The `str` class provides many member functions. Since string is immutable, most of these functions return a new string. The commonly-used member functions are as follows, supposing that `s` is a `str` object:

- `s.strip()`, `s.rstrip()`, `s.lstrip()`: the `strip()` strips the leading and trailing whitespaces. The `rstrip()` strips the right (trailing) whitespaces; while `lstrip()` strips the left (leading) whitespaces.
- `s.upper()`, `s.lower()`
- `s.isupper()`, `s.islower()`
- `s.find()`
- `s.index()`
- `s.startswith()`
- `s.endswith()`
- `s.split(delimiter-str)`, `delimiter-str.join(list-of-strings)`
- `s.count()`

```
>>> dir(str)      # List all attributes of the class str
......
>>> s = 'Hello, world'
>>> type(s)
<class 'str'>

>>> s.find('ll')   # Find the beginning index of the substring
2
>>> s.find('app')  # find() returns -1 if not found
-1

>>> s.index('ll')  # index() is the same as find(), but raise ValueError if not found
2
>>> s.index('app')
......
ValueError: substring not found

>>> s.startswith('Hell')
True
>>> s.endswith('world')
True
>>> s.replace('ll', 'xxx')
'Hexxxo, world'
>>> s.isupper()
False
>>> s.upper()
'HELLO, WORLD'
```

```
>>> s.split(', ')    # Split into a list with the given delimiter
['Hello', 'world']
>>> ', '.join(['hello', 'world', '123'])  # Join all strings in the list using the delimiter
'hello, world, 123'

>>> s = '  testing testing   '
>>> s.strip()        # Strip leading and trailing whitespaces
'testing testing'
>>> s.rstrip()       # Strip trailing (right) whitespaces
'  testing testing'
>>> s.lstrip()       # Strip leading (left) whitespaces
'testing testing   '
>>> s.count('o')       # Count the number of occurrence of 'o'
2
```

String Formatting: Using `str.format()` function

Python 3 introduces a new style in the string's `format()` function with `{}` as place-holders (called format fields). For examples,

```
# Replace format fields {} by arguments in format() in the same order
>>> '|{}|{}|more|'.format('Hello', 'world')
'|Hello|world|more|'

# You can use positional index in the form of {0}, {1}, ...
>>> '|{0}|{1}|more|'.format('Hello', 'world')
'|Hello|world|more|'
>>> '|{1}|{0}|more|'.format('Hello', 'world')
'|world|Hello|more|'
```

String Formatting 3: Using `str.rjust(n)`, `str.ljust(n)`, `str.center(n)`, `str.zfill(n)`

You can also use string's functions like `str.rjust(n)` (where `n` is the field-width), `str.ljust(n)`, `str.center(n)`, `str.zfill(n)` to format a string. For example,

```
# Setting field width and alignment
>>> '123'.rjust(5)
'  123'
>>> '123'.ljust(5)
'123  '
>>> '123'.center(5)
' 123 '
>>> '123'.zfill(5)  # Pad with leading zeros
```

```
'00123'

# Floats
>>> '1.2'.rjust(5)
'  1.2'
>>> '-1.2'.zfill(6)
'-001.2'
```

Conversion between String and Number: `int()`, `float()` and `str()`

You can use built-in functions `int()` and `float()` to parse a "numeric" string to an integer or a float; and `str()` to convert a number to a string. For example,

```
>>> s = '12345'
>>> s
'12345'
>>> int(s)    # Convert string to int
12345
>>> s = '55.66'
>>> s
'55.66'
>>> float(s)  # Convert string to float
55.66
>>> int(s)
ValueError: invalid literal for int() with base 10: '55.66'
>>> i = 8888
>>> str(i)    # Convert number to string
'8888'
```

Concatenate a String and a Number?

You CANNOT concatenate a string and a number (which results in `TypeError`). Instead, you need to use the `str()` function to convert the number to a string. For example,

```
>>> 'Hello' + 123
TypeError: cannot concatenate 'str' and 'int' objects
>>> 'Hello' + str(123)
'Hello123'
```

The `isinstance()` Built-in Function

You can use the built-in function `isinstance(instance, type)` to check if the `instance` belong to the `type`. For example,

```
>>> isinstance(123, int)
True
>>> isinstance('a', int)
False
>>> isinstance('a', str)
True
```

6.6 List `[v1, v2,...]`

Python has a powerful built-in list for *dynamic array*.

- A list is enclosed by square brackets `[]`.
- A list can contain items of different types.
- A list grows and shrinks in size automatically (dynamically). You do not have to specify its size during initialization.

Built-in Functions and Operators for Lists

A list, like string, is a sequence. Hence, you can operate lists using:

- built-in sequence functions such as `len()`.
- built-in sequence functions for list of numbers such as `max()`, `min()`, and `sum()`.
- operators such as `in` (contains), `+` (concatenation) and `*` (repetition), `del`, indexing `[i]`, and slicing `[m,n,step]`.

Notes:

- You can index the items from the front with positive index, or from the back with negative index. E.g., if `x` is a list, `x[0]` and `x[1]` refer to its first and second items; `x[-1]` and `x[-2]` refer to the last and second-to-last items.
- You can also refer to a sub-list (or slice) using slice notation `x[m:n]` (from index `m` (included) to index `n` (excluded)), `x[m:]` (to last item), `x[:n]` (from first item), `x[:]` (all items), and `x[m,n,step]`(in step size).

Operator	Description	Examples `lst = [8, 9, 6, 2]`

`in`	Contain?	`9 in lst ⇒ True`
		`5 in lst ⇒ False`

`+`	Concatenation	`lst + [5, 2]`
		`⇒ [8, 9, 6, 2, 5, 2]`

`*`	Repetition	`lst * 2`
		`⇒ [8, 9, 6, 2, 8, 9, 6, 2]`

`[i], [-i]`	Indexing to get an item.
	Front index begins at `0`; back index begins at `-1` (or `len-1`).
		`lst[1] ⇒ 9`
		`lst[-2] ⇒ 6`
		`lst[1] = 99 ⇒ modify an existing item`

`[m:n],`
`[m:], [:n],`
`[m:n:step]`
	Slicing to get a sublist.
	From index `m` (included) to `n` (excluded) with an optional `step`size.
	The default `m` is `0`, `n` is `len-1`.
		`lst[1:3] ⇒ [9, 6]`
		`lst[1:-2] ⇒ [9]`
		`lst[3:] ⇒ [2]`
		`lst[:-2] ⇒ [8, 9]`
		`lst[:] ⇒ [8, 9, 6, 2]`
		`lst[0:4:2] ⇒ [8, 6]`
		`newlst = lst[:] ⇒ copy the list`
		`lst[4:] = [1, 2] ⇒ modify a sub-list`

`del`	Delete one or more items
	(for mutable sequences only)
		`del lst[1] ⇒ lst is [8, 6, 2]`
		`del lst[1:] ⇒ lst is [8]`
		`del lst[:] ⇒ lst is [] (clear all items)`

Function	Description	Examples `lst = [8, 9, 6, 2]`
`len()`	Length	`len(lst) ⇒ 4`

`max(),`
`min()`
	Maximum and minimum value (for list of numbers only)	`max(lst) ⇒ 9`
		`min(lst) ⇒ 2`

`sum()`	Sum (for list of numbers only)	`sum(lst) ⇒ 16`

List, unlike string, is mutable. You can insert, remove and modify its items.

For examples,

```
>>> lst = [123, 4.5, 'hello']  # A list can contains items of different types
>>> lst
[123, 4.5, 'hello']
>>> len(lst)   # Length
3
>>> type(lst)
```

```
<class 'list'>
>>> lst[0]     # Indexing to get an item
123
>>> lst[2] = 'world'  # Re-assign
>>> lst
[123, 4.5, 'world']
>>> lst[0:2]   # Slicing to get a sub-list
[123, 4.5]
>>> lst[:2]
[123, 4.5]
>>> lst[1:]
[4.5, 'world']
>>> 123 in lst
True
>>> 1234 in lst
False
>>> lst + [6, 7, 8]    # Concatenation
[123, 4.5, 'world', 6, 7, 8]
>>> lst * 3            # Repetition
[123, 4.5, 'world', 123, 4.5, 'world', 123, 4.5, 'world']
>>> del lst[1]         # Removal
>>> lst
[123, 'world']

# Lists can be nested
>>> lst = [123, 4.5, ['a', 'b', 'c']]
>>> lst
[123, 4.5, ['a', 'b', 'c']]
>>> lst[2]
['a', 'b', 'c']
```

Appending Items to a list

```
>>> lst = [123, 'world']
>>> lst[2]     # Python performs index bound check
IndexError: list index out of range
>>> lst.append('nine')  # Append an item via append() function
>>> lst

[123, 'world', 4.5, 6, 7, 8, 'nine']
```

```
>>> lst.extend(['a', 'b'])  # extend() takes a list
>>> lst
[123, 'world', 4.5, 6, 7, 8, 'nine', 'a', 'b']

>>> lst + ['c']  # '+' returns a new list;
[123, 'world', 4.5, 6, 7, 8, 'nine', 'a', 'b', 'c']
>>> lst  # No change
[123, 'world', 4.5, 6, 7, 8, 'nine', 'a', 'b']
```

Copying a List

```
>>> l1 = [123, 4.5, 'hello']
>>> l2 = l1[:]   # Make a copy via slicing
>>> l2
[123, 4.5, 'hello']
>>> l2[0] = 8    # Modify new copy
>>> l2
[8, 4.5, 'hello']
>>> l1           # No change in original
[123, 4.5, 'hello']

>>> l3 = l1.copy()   # Make a copy via copy() function, same as above

# Contrast with direct assignment
>>> l4 = l1    # Direct assignment (of reference)
>>> l4
[123, 4.5, 'hello']
>>> l4[0] = 8  # Modify new copy
>>> l4
[8, 4.5, 'hello']
>>> l1         # Original also changes
[8, 4.5, 'hello']
```

List-Specific Member Functions

The `list` class provides many member functions. Suppose `lst` is a `list` object:

- `lst.append(item)`: append the given `item` behind the `lst` and return `None`; same as `lst[len(lst):] = [item]`.
- `lst.extend(lst2)`: append the given list `lst2` behind the `lst` and return `None`; same as `lst[len(lst):] = lst2`.

- `lst.insert(index, item)`: insert the given `item` before the `index` and return `None`. Hence, `lst.insert(0, item)` inserts before the first item of the `lst`; `lst.insert(len(lst), item)`inserts at the end of the `lst` which is the same as `lst.append(item)`.
- `lst.index(item)`: return the index of the first occurrence of `item`; or error.
- `lst.remove(item)`: remove the first occurrence of `item` from the `lst` and return `None`; or error.
- `lst.pop()`: remove and return the last item of the `lst`.
- `lst.pop(index)`: remove and return the indexed item of the `lst`.
- `lst.clear()`: remove all the items from the `lst` and return `None`; same as `del lst[:]`.
- `lst.count(item)`: return the occurrences of `item`.
- `lst.reverse()`: reverse the `lst` in place and return `None`.
- `lst.sort()`: sort the `lst` in place and return `None`.
- `lst.copy()`: return a copy of `lst`; same as `lst[:]`.

Recall that list is mutable (unlike string which is immutable). These functions modify the list directly. For examples,

```
>>> lst = [123, 4.5, 'hello', [6, 7, 8]]  # list can also contain list
>>> lst
[123, 4.5, 'hello', [6, 7, 8]]
>>> type(lst)  # Show type
<class 'list'>
>>> len(lst)
4
>>> lst.append('apple')  # Append item at the back
>>> lst
[123, 4.5, 'hello', [6, 7, 8], 'apple']
>>> len(lst)
5
>>> lst.pop(1)     # Retrieve and remove item at index
4.5
>>> lst
[123, 'hello', [6, 7, 8], 'apple']
>>> len(lst)
4
>>> lst.insert(2, 55.66)  # Insert item before the index
>>> lst
[123, 'hello', 55.66, [6, 7, 8], 'apple']
>>> del lst[3:]         # Delete the slice (del is an operator , not function)


>>> lst
```

```
[123, 'hello', 55.66]
>>> lst.append(55.66)   # A list can contain duplicate values
>>> lst
[123, 'hello', 55.66, 55.66]
>>> lst.remove(55.66)   # Remove the first item of given value
>>> lst
[123, 'hello', 55.66]
>>> lst.reverse()       # Reverse the list in place
>>> lst
[55.66, 'hello', 123]

# Searching and Sorting
>>> lst2 = [5, 8, 2, 4, 1]
>>> lst2.sort()     # In-place sorting
>>> lst2
[1, 2, 4, 5, 8]
>>> lst2.index(5)   # Get the index of the given item
3
>>> lst2.index(9)
......
ValueError: 9 is not in list
>>> lst2.append(1)
>>> lst2
[1, 2, 4, 5, 8, 1]
>>> lst2.count(1)   # Count the occurrences of the given item
2
>>> lst2.count(9)
0
>>> sorted(lst2)    # Built-in function that returns a sorted list
[1, 1, 2, 4, 5, 8]
>>> lst2
[1, 2, 4, 5, 8, 1]  # Not modified
```

6.7 Tuple (v1, v2,...)

Tuple is similar to list except that it is immutable (just like string). A tuple consists of items separated by commas, enclosed in parentheses ().

```
>>> tup = (123, 4.5, 'hello')  # A tuple can contain different types
>>> tup
(123, 4.5, 'hello')
>>> tup[1]          # Indexing to get an item
4.5
>>> tup[1:3]        # Slicing to get a sub-tuple
(4.5, 'hello')
>>> tup[1] = 9      # Tuple, unlike list, is immutable
TypeError: 'tuple' object does not support item assignment
>>> type(tup)
<class 'tuple'>
>>> lst = list(tup)  # Convert to list
>>> lst
[123, 4.5, 'hello']
>>> type(lst)
<class 'list'>
```

An one-item tuple needs a comma to differentiate from parentheses:

```
>>> tup = (5,)  # An one-item tuple needs a comma
>>> tup
(5,)
>>> x = (5)     # Treated as parentheses without comma
>>> x
5
```

The parentheses are actually optional, but recommended for readability. Nevertheless, the commas are mandatory. For example,

```
>>> tup = 123, 4.5, 'hello'
>>> tup
(123, 4.5, 'hello')
>>> tup2 = 88,  # one-item tuple needs a trailing commas
>>> tup2
(88,)
```

```
# However, we can use empty parentheses to create an empty tuple
# Empty tuples are quite useless, as tuples are immutable.
>>> tup3 = ()
>>> tup3
()
>>> len(tup3)
0
```

You can operate on tuples using (supposing that `tup` is a tuple):

- built-in functions such as `len(tup)`;
- built-in functions for tuple of numbers such as `max(tup)`, `min(tup)` and `sum(tup)`;
- operators such as `in`, `+` and `*`; and
- `tuple`'s member functions such as `tup.count(item)`, `tup.index(item)`, etc.

Conversion between List and Tuple

You can covert a list to a tuple using built-in function `tuple()`; and a tuple to a list using `list()`. For examples,

```
>>> tuple([1, 2, 3, 1])  # Convert a list to a tuple
(1, 2, 3, 1)
>>> list((1, 2, 3, 1))   # Convert a tuple to a list
[1, 2, 3, 1]
```

6.8 Dictionary `{k1:v1, k2:v2,...}`

Python's built-in dictionary type supports *key-value pairs* (also known as *name-value pairs*, *associative array*, or *mappings*).

- A dictionary is enclosed by a pair of curly braces `{}`. The key and value are separated by a colon (`:`), in the form of `{k1:v1, k2:v2, ...}`
- Unlike list and tuple, which index items using an integer index (0, 1, 2, 3,...), dictionary can be indexed using any key type, including number, string or other types.

```
>>> dct = {'name':'Peter', 'gender':'male', 'age':21}
>>> dct
{'age': 21, 'name': 'Peter', 'gender': 'male'}
>>> dct['name']       # Get value via key
'Peter'
>>> dct['age'] = 22   # Re-assign a value
>>> dct
{'age': 22, 'name': 'Peter', 'gender': 'male'}
>>> len(dct)
3
>>> dct['email'] = 'peter@nowhere.com'   # Add new item
>>> dct
{'name': 'Peter', 'age': 22, 'email': 'peter@nowhere.com', 'gender': 'male'}
>>> type(dct)
<class 'dict'>

# Use dict() built-in function to create a dictionary
>>> dct2 = dict([('a', 1), ('c', 3), ('b', 2)])  # Convert a list of 2-item tuples into a dictionary
>>> dct2
{'b': 2, 'c': 3, 'a': 1}
```

Dictionary-Specific Member Functions

The `dict` class has many member methods. The commonly-used are follows (suppose that `dct` is a `dict` object):

- `dct.has_key()`:
- `dct.items()`, `dct.keys()`, `dct.values()`:
- `dct.clear()`:
- `dct.copy()`:
- `dct.get()`:

- `dct.update(dct2)`: merge the given dictionary `dct2` into `dct`. Override the value if key exists, else, add new key-value.
- `dct.pop()`:

For Examples,

```
>>> dct = {'name':'Peter', 'age':22, 'gender':'male'}
>>> dct
{'gender': 'male', 'name': 'Peter', 'age': 22}

>>> type(dct)  # Show type
<class 'dict'>
>>> list(dct.keys())       # Get all the keys as a list
['gender', 'name', 'age']
>>> list(dct.values())     # Get all the values as a list
['male', 'Peter', 22]
>>> list(dct.items())      # Get key-value as tuples
[('gender', 'male'), ('name', 'Peter'), ('age', 22)]

# You can also use get() to retrieve the value of a given key
>>> dct.get('age')  # Retrieve item
22
>>> dct['height']
KeyError: 'height'
    # Indexing an invalid key raises KeyError, while get() could gracefully handle invalid key

>>> del dct['age']   # Delete (Remove) an item of the given key
>>> dct
{'gender': 'male', 'name': 'Peter'}

>>> 'name' in dct
True

>>> dct.update({'height':180, 'weight':75})  # Merge the given dictionary
>>> dct
{'gender': 'male', 'name': 'Peter', 'height': 180, 'weight': 75}

>>> dct.pop('gender')  # Remove and return the item with the given key
'male'
>>> dct
{'name': 'Peter', 'weight': 75, 'height': 180}
```

6.9 Set {k1, k2,...}

A set is an unordered, non-duplicate collection of objects. A set is delimited by curly braces {}, just like dictionary. You can think of a set as a collection of dictionary keys without associated values.

For example,

```
>>> st = {123, 4.5, 'hello', 123, 'Hello'}
>>> st         # Duplicate removed and ordering may change
{'Hello', 'hello', 123, 4.5}
>>> 123 in st  # Test membership
True
>>> 88 in st
False

# Use the built-in function set() to create a set.
>>> st2 = set([2, 1, 3, 1, 3, 2])  # Convert a list to a set. Duplicate removed and unordered.
>>> st2
{1, 2, 3}
>>> st3 = set('hellllo')  # Convert a string to a character set.
>>> st3
{'o', 'h', 'e', 'l'}
```

Set-Specific Operators

Python supports set operators & (intersection), | (union), - (difference) and ^ (exclusive-or). For example,

```
>>> st1 = {'a', 'e', 'i', 'o', 'u'}
>>> st1
{'e', 'o', 'u', 'a', 'i'}
>>> st2 = set('hello')  # Convert a string to a character set
>>> st2
{'o', 'l', 'e', 'h'}
>>> st1 & st2   # Set intersection
{'o', 'e'}
>>> st1 | st2   # Set union
{'o', 'l', 'h', 'i', 'e', 'a', 'u'}
>>> st1 - st2   # Set difference
{'i', 'u', 'a'}

>>> st1 ^ st2   # Set exclusive-or
```

```
{'h', 'i', 'u', 'a', 'l'}
```

7. Control Structures

7.1 Syntax

A Python compound statement (such as conditional, loop and function definition) takes the following syntax:

```
Header-1:        # Headers are terminated by a colon
    statement-1  # Body blocks are indented (recommended to use 4 spaces)
    statement-2
    ......
Header-2:
    statement-1
    statement-2
    ......
```

For example, a `if-elif-else` statement takes the following form:

```
if x == 0:    # Parentheses are optional for condition
    print('x is zero')
elif x > 0:
    print(x)
    print('x is more than zero')
else:
    print(x)
    print('x is less than zero')
```

- It begins with a header line, such as `if`, `elif`, `else`, terminated by a colon (`:`), followed by the *indented* body block of statements.
- The parentheses `()` surrounding the condition are optional.
- Python does not use curly braces `{}` to embrace a block (as in C/C++/Java). Instead, it uses *indentation* to delimit a body block. That is, the indentation is syntactically significant. Python does not care how you indent (with spaces or tab), and how far you indent. But ALL the statements in the SAME block must be indented with the SAME distance. You can use either spaces or tabs, but you cannot mix them. It is recommended to use 4 spaces for the indentation. This indentation rule forces programmers to write readable codes!!!

In Python, a statement is delimited by an end-of-line. That is, the end-of-line is the end of the statement. There is no need to place a semicolon (`;`) to end a statement (as in C/C++/Java).

Special cases:

- You can place multiple statements in one line, separated by semicolon (`;`), for example,

```
x = 1; y = 2; z = 3
```

- You can place the body block on the same line as the header line, e.g.,

```
if x == 0:   print('x is zero')
elif x > 0:  print(x); print('x is more than zero')
else:        print(x); print('x is less than zero')
```

7.2 Conditional `if-elif-else`

The general syntax is as follows. The `elif` (else-if) and `else` blocks are optional.

```
if test-1:
    block-1
elif test-2:
    block-2
elif test-3:
    block-3
......
elif test-n:
    block-n
else:
    else-block
```

There is no `switch-case` statement in Python (as in C/C++/Java).

Comparison and Logical Operators

Python supports these comparison (relational) operators, which return a `bool` of either `True` or `False`

- `<` (less than), `<=` (less than or equal to), `==` (equal to), `!=` (not equal to), `>` (greater than), `>=` (gr. than or equal to).
- `in`, `not in`: Check if an item is|is not in a sequence (list, tuple, etc).
- `is`, `is not`: Check if two variables have the same reference.

Python supports these logical (boolean) operators:

- `and`
- `or`
- `not`

Chain Comparison `n1 < x < n2`

Python supports chain comparison in the form of `n1 < x < n2` (which is not supported in C/C++/Java), e.g.,

```
>>> x = 8
>>> 1 < x < 10
True
>>> 1 < x and x < 10  # Same as above
True
>>> 10 < x < 20
False
>>> 10 > x > 1
True
>>> not (10 < x < 20)
True
```

Comparing Sequences

The comparison operators (such as ==, <=) are overloaded to support sequences (such as string, list and tuple). For example,

```
>>> x = 8
>>> 'a' < 'b'
True
>>> 'ab' < 'aa'
False
>>> 'a' < 'b' < 'c'
True
>>> (1, 2, 3) < (1, 2, 4)
True
>>> [1, 2, 3] <= [1, 2, 3]
True
```

Shorthand if-else

The syntax is:

```
expr-1 if test else expr-2
    # Evaluate expr-1 if test is True; otherwise, evaluate expr-2
```

For example,

```
>>> x = 0
>>> print('zero' if x == 0 else 'not zero')
zero
```

Note: Python does not use "? :" for shorthand if-else, as in C/C++/Java.