AI전문가과정 - Python Review

Structured Types in Python

Jinwook Seo, Ph.D.
Professor, Department of Computer Science and Engineering
Seoul National University

Table of Contents

AI전문가과정 - Python Review

Object

Data Model of Python

- Objects are Python's abstraction for data.
- All data in a Python program is represented by objects or by relations between objects.
  - code is also represented by objects
- Every object has an identity, a type and a value.

Identity, Type, Value of Objects

- Identity
  - An object's identity never changes once it has been created
  - you may think of it as the object's address in memory
  - `id()` function returns an integer representing its identity
  - `is` operator compares the identity of two objects
- Type
  - An object's type determines the operations that the object supports
  - Defines the possible values for objects of that type
  - `type()` function returns an object's type (type itself is an object)
  - an object's type is also unchangeable.
- Value
  - The value of some objects can change
  - Objects whose value can change are said to be `mutable`
  - Objects whose value is unchangeable once they are created are called `immutable`

Objects in Python

- Objects are never explicitly destroyed
  - however, when they become unreachable they may be garbage-collected
- Some objects contain references to “external” resources such as open `files` or `windows`
- Some objects contain references to other objects; these are called containers
  - e.g., `tuples`, `lists` and `dictionaries`

Objects in Python

- Types affect almost all aspects of object behavior
- Even object identity is affected by its type in some sense

- for immutable types, operations that compute new values may actually return a reference to an existing object with the same type and value (to save memory)
  - after `a = 1; b = 1`, `a` and `b` may or may not refer to the same object with the value one, depending on the implementation
  - (Note that `a = b = 1` always assigns the same object to both `a` and `b`.)

```
>>> a = 1
>>> b = 1
>>> a is b
True
>>> a = 2000
>>> b = 2000
>>> a is b
False
>>> a = b = 2000
>>> a is b
True
```

```
>>> word="string"
>>> word1="string"
>>> word is word1 # string is immutable
True
>>> s='14123412341234123412341234afasdf'
>>> s1='14123412341234123412341234afasdf'
>>> s1 is s
True
```

Objects in Python

- Types affect almost all aspects of object behavior
- Even object identity is affected by its type in some sense

- for mutable objects, it is NOT allowed for operations that compute new values to return a reference to any existing object
  - after `c = []; d = []`, `c` and `d` are guaranteed to refer to two different, unique, newly created empty lists
  - (Note that `c = d = []` assigns the same object to both `c` and `d`.)

```
>>> c = []
>>> d = []
>>> c is d
False
>>> c = d = []
>>> c is d
True
```

The Standard Type Hierarchy

- Built-in Types: the types that are built into Python

```
None
NotImplemented
Ellipsis
numbers.Number
    numbers.Integral
        Integers(int)
        Booleans(bool)
    numbers.Real(float)
    numbers.Complex(complex)
Sequences
    Immutable sequences
        Strings
        Tuples
        Bytes
    Mutable sequences
        Lists
        Byte Arrays
```

```
Set types
    Sets
    Frozen sets
Mappings
    Dictionaries
Callable types
    User-defined functions
    Instance methods
    Generator functions
    Asynchronous generator functions
    Coroutine functions
    Classes, Class instances
Custum classes
Class instances
I/O objects
Internal types
```

Structured Types in Python

- Objects of structured types have an accessible internal structure
  - v.s. scalar types: `int`, `float`
- sequence types are structured types:
  - `string`, `list`, `tuple`, (`buffer`)
  - sequence operations sorted in ascending priority

Operation	Result
`x in s`	`True` if an item of *s* is equal to *x*, else `False`
`x not in s`	`False` if an item of *s* is equal to *x*, else `True`
`s + t`	the concatenation of *s* and *t*
`s * n , n * s`	*n* shallow copies of *s* concatenated
`s[i]`	*i*'th item of *s*, origin 0
`s[i:j]`	slice of *s* from *i* to *j*
`s[i:j:k]`	slice of *s* from *i* to *j* with step *k*
`len(s)`	length of *s*
`min(s)`	smallest item of *s*
`max(s)`	largest item of *s*

`s` and `t` are sequences of the same type; `n`, `i` and `j` are integers

AI전문가과정 - Python Review

String

String: Text Sequence Type

- immutable
- Indexing, Concatenation, Repetition

```
>>> word = 'Python'
>>> word[0]  # character in position 0
'P'
>>> word[5]  # character in position 5
'n'
>>> word[-1]  # last character
'n'
>>> word[-6]
'P'
>>> word[42]  # the word only has 6 characters
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>> 'Python'[1]
'y'
```

```
# the position of the indices
 +---+---+---+---+---+---+
 | P | y | t | h | o | n |
 +---+---+---+---+---+---+
 0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

```
>>> 'Py'  'thon' # concatenation
'Python'
>>> word + '3'   # concatenation
'Python3'
>>> word * 3     # repetition
'PythonPythonPython'
```

Iterating over a Sequence

- `in`, `not in`, iteration

```
>>> word = 'Python'
>>> 'P' in 'Python'
True
>>> 'p' not in word
True
>>> for c in word:
...     print(c, end='/')


P/y/t/h/o/n/
```

Slicing

- the start is always `included`, and the end is always `excluded`
- slice indices have useful defaults;
  - an omitted first index defaults to `zero`,
  - an omitted second index defaults to the `size of the string` being sliced

```
>>> word = 'Python'
>>> word[0:2]  # characters from position 0 (included) to 2 (excluded)
'Py'
>>> word[2:5]  # characters from position 2 (included) to 5 (excluded)
'tho'
>>> word[:2] + word[2:]
'Python'
>>> word[:4] + word[4:]
'Python'
>>> word[:2]   # character from the beginning to position 2 (excluded)
'Py'
>>> word[4:]   # characters from position 4 (included) to the end
'on'
>>> word[-2:]  # characters from the second-last (included) to the end
'on'
```

Slicing

- out of range slice indexes are handled gracefully when used for slicing:

```
>>> word[4:42]
'on'
>>> word[42:]
''
>>> numbers = '0123456789'
>>> numbers[0:10:2]  # an omitted third value (i.e., step) defaults to 1
'02468'
>>> numbers[1:10:2]
'13579'
```

Slicing

- Python strings cannot be changed — they are immutable

```
>>> word[0] = 'J'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> word[2:] = 'py'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>>
>>>
>>> jython = 'J' + word[1:]
>>> jython
'Jython'
>>> word[:2] + 'py'
'Pypy'
```

Some Methods on Strings

- str.`count`(s1)
- str.`find`(s1)
- str.`rfind`(s1)
- str.`index`(s1) # raise an exception if `s1` is not in `str`
- str.`rindex`(s1) # raise an exception if `s1` is not in `str`
- str.`lower`()
- str.`replace`(old, new)
- str.`rstrip`()
- str.`strip`()
- str.`split`(d)
  - return a `list` with strings after splitting str using a delimiter, `d`
  - when `d` is omitted, white space is used as a delimiter
- for more information:
  - https://docs.python.org/3/library/stdtypes.html#str

AI전문가과정 - Python Review

Tuple

Tuples

- Like `strings`, `tuples` are immutable ordered sequences of elements
- The elements of a `tuple` can be of any type
- The elements need not be of the same type as each other

- Literals of type `tuple`

```
>>> t1 = ()  # empty tuple
>>> t2 = (1, 'two', 3.0)
>>> print(t1)
()
>>> print(t2)
(1, 'two', 3.0)
>>>
>>> t3 = (1)  # (1) is the same as integer 1
>>> t4 = (1,) # singleton tuple with integer 1
>>> print(t3)
1
>>> print(t4)
(1,)
```

Repetition, Concatenation, Indexing, and Slicing

```
t1 = 3 * ('1', 2)
t2 = (1, 'two', 3.0)
t3 = (t2, 3.35)
print(t1)            # prints ('1', 2, '1', 2, '1', 2)
print(t3)            # prints ((1, 'two', 3.0), 3.35)
print(t2 + t3)       # prints (1, 'two', 3.0, (1, 'two', 3.0), 3.35)
print((t2 + t3)[3])  # prints (1, 'two', 3.0)
print((t2 + t3)[2:5] # prints (3.0, (1, 'two', 3.0), 3.35)
```

Iterate over a tuple

```
def intersect(t1, t2):
    """
    Assumes t1 and t2 are tuples
    Returns a tuple containing elements that are in both t1 and t2
    """
    result = ()
    for e in t1:
        if e in t2:
            result += (e,)  # result references a new tuple object
    return result

result = intersect((1, 2, 3), (2, 3, 4))
print(result)  # prints (2, 3)
```

Run Code Visualize

Sequences, Unpacking, and Multiple Assignment

```
x, y = (3, 4)    # unpacking
print(x, y)      # prints 3 4
x, *y = (3, 4, 5, 6)
print(x, y)      # prints 3 [4, 5, 6]
*x, y = (3, 4, 5, 6)
print(x, y)      # prints [3, 4, 5] 6
a, b, c = 'xyz'  # unpacking
print(a, b, c)   # prints x y z
a, *b = 'xyz'    # unpacking
print(a, b)      # prints x ['y', 'z']
```

Run Code Visualize

Returning Multiple Objects as a Tuple

```
def sum_and_mul(a, b):
    return a + b, a * b   # returns a tuple

result = sum_and_mul(1, 2)
print(result)  # prints (3, 2)

sum, mul = sum_and_mul(1, 2)  # unpacking the tuple into two objects
print(sum)     # prints 3
print(mul)     # prints 2
```

Run Code Visualize

Ranges

- Like `strings` and `tuples`, ranges are immutable
- The built-in function `range` returns an object of type `range`
- All of the operations on `tuples` are also available for ranges
    - but contatenation and repetition are not

```
for num in range(5):
    print(num, end=' ')
# prints 0 1 2 3 4

for num in range(1, 6):
    print(num, end=' ')
# prints 1 2 3 4 5



for num in range(1, 10, 2):
    print(num, end=' ')
# prints 1 3 5 7 9
```

Ranges

- two range objects are equal if they represent the same sequence of integers

```
>>> range(0, 7, 2) == range(0, 8, 2)
True
>>> range(0, 7, 2) == range(6, -1, -2)  # the order of elements matters
False
```

- amount of memory space occupied by a range is not propotional to its length
- but it is for a `tuple`/`string`/`list`

AI전문가과정 - Python Review

List

Lists

- Python has a number of compound data types, used to group together other values (possibily of different types)
- Some objects contain references to other objects; these are called containers
  - e.g., `tuple`, `list` and `dictionary`
- The most versatile is the `list`
  - written as a list of comma-separated values (items) between square brackets

```
>>> squares = [1, 4, 9, 16, 25]
>>> squares
[1, 4, 9, 16, 25]
>>> squares[0]  # indexing returns the item at the given index
1
>>> squares[-1]
25
>>> one2ten = list(range(1, 11))
>>> one2ten
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Lists are `Mutable`

- lists are a `mutable` type, i.e. it is possible to change their content

```
>>> cubes = [1, 8, 27, 65, 125]
>>> 4 ** 3  # the cube of 4 is 64, not 65!
64
>>> cubes[3] = 64  # replace the wrong value
>>> cubes
[1, 8, 27, 64, 125]
>>>
>>> cubes.append(216)  # add the cube of 6
>>> cubes.append(7 ** 3)  # and the cube of 7
>>> cubes
[1, 8, 27, 64, 125, 216, 343]
```

Slicing and Concatenation

- All slice operations return a new list containing the requested elements
  - note: slice on the left side of assignment mutates the original list

Nested Lists

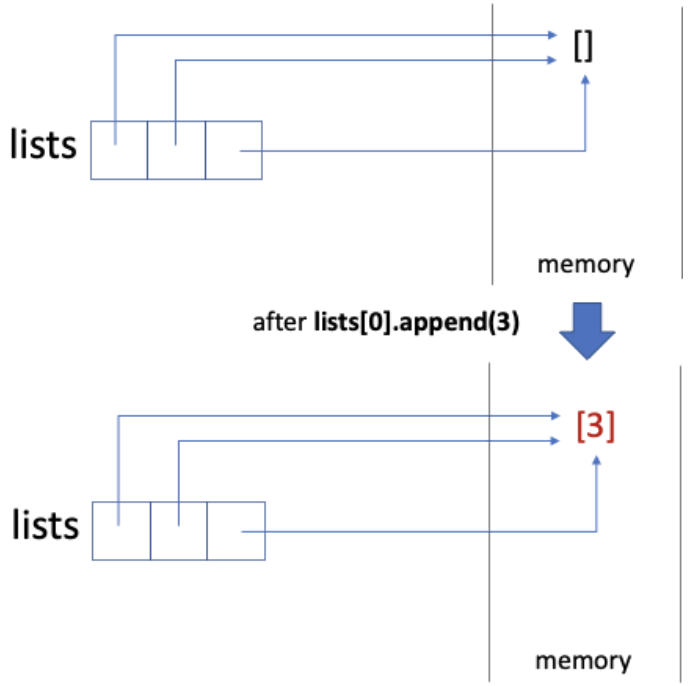
- It is possible to nest lists (create a list than contains other lists)

```
>>> a = ['a', 'b', 'c']
>>> n = [1, 2, 3]
>>> x = [a, n]
>>> x
[['a', 'b', 'c'], [1, 2, 3]]
>>> x[0]
['a', 'b', 'c']
>>> x[0][1]
'b'
```

Repetition: Returns Shallow Copy of a List

- `Repetition` returns a concatenated list of shallow copies of a list:

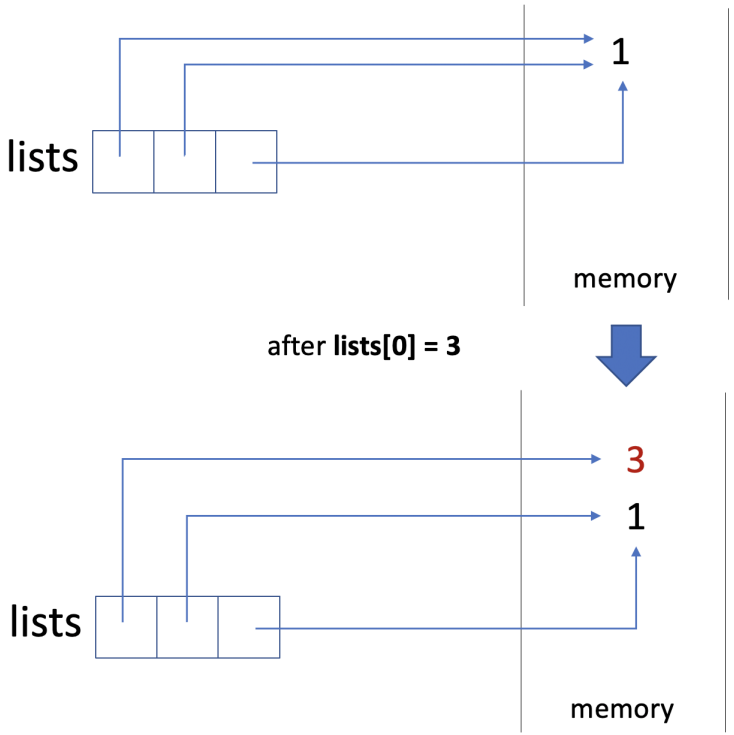
```
>>> lists = [[]] * 3
# 3 shallow copies of [[]] are concatenated!
>>> lists
[[], [], []]
>>> lists[0].append(3)
>>> lists
[[3], [3], [3]]
>>> lists[1].append(4)
>>> lists
[[3, 4], [3, 4], [3, 4]]
```



Repetition: Returns Shallow Copy of a List

- `Repetition` returns a concatenated list of shallow copy of a list:

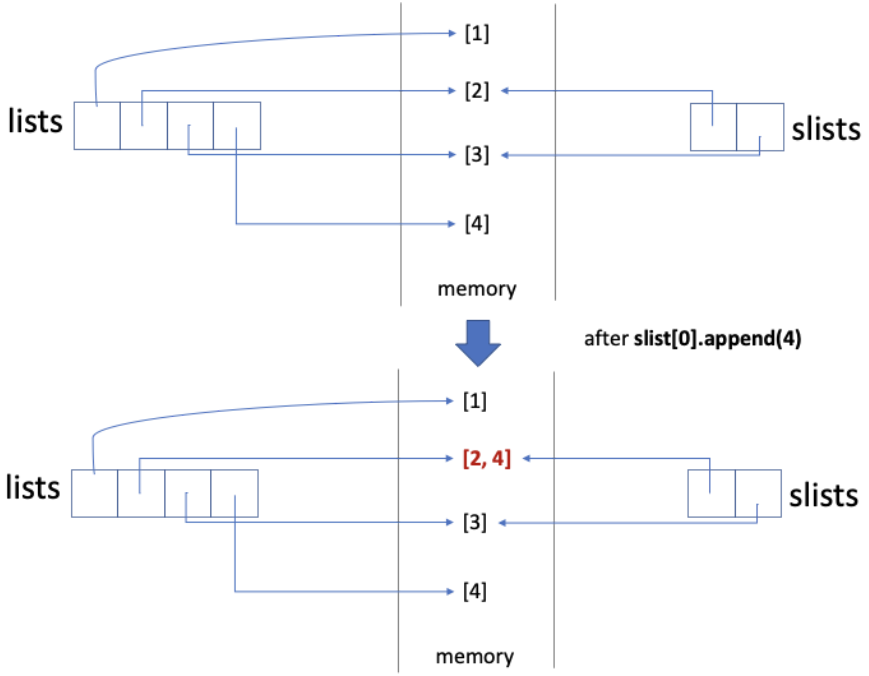
```
>>> lists = [1] * 3
>>> lists
[1, 1, 1]
>>> lists[0] = 3
>>> lists
[3, 1, 1]
>>> lists[1]=4
>>> lists
[3, 4, 1]
```



Slicing: Returns Shallow Copy of a List

- `Slicing` returns a shallow copy of the list:

```
>>> lists = [[1], [2], [3], [4]]
>>> slists = lists[1:3]
>>> slists
[[2], [3]]
>>> slist[0].append(4)
>>> slists
[[2, 4], [3]]
>>> lists
[[1], [2, 4], [3], [4]]
>>>
```



Assignment to Slices

- Assignment to slices is also possible
  - when slice is on the left side of assignment
  - note: slice on the right side returns a new list
- The original list is mutated

```
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> letters
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> # replace some values
>>> letters[2:5] = ['C', 'D', 'E']  # letters is mutated!
>>> letters
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>> # now remove them
>>> letters[2:5] = []
>>> letters
['a', 'b', 'f', 'g']
>>> # clear the list by replacing all the elements with an empty list
>>> letters[:] = []
>>> letters
[]
```

Lists and Mutability

```
>>> Techs = ['KAIST', 'Postech']
>>> Sky = ['SNU', 'Korea', 'Yonsei']
>>> Univs = [Sky, Techs]
>>> Univs1 = [['SNU', 'Korea', 'Yonsei'], ['KAIST', 'Postech']]
>>> Univs == Univs1  # test value equality
True
>>> id(Univs) == id(Univs1) # test equality of object identities
False
>>> Univs is Univs1 # test object identity
False
>>> Techs.append('GIST')
>>> Univs
[['SNU', 'Korea', 'Yonsei'], ['KAIST', 'Postech', 'GIST']]
```

- aliasing: happens when there are two distinct paths to the same object
  - `alias` is another name for an object

Side Effect

- The `append` method has a side effect:
  - It doesn't return anything (no main effect), but mutates the existing list
- Function or expression is said to have a side effect
  - if it modifies some variable value(s) outside its local environment
  - if it makes an observable effect besides returning a value (the main effect) to the invoker of the operation
- Examples
  - modifying a non-local variable (global variable, or variable from outer scope)
  - (modifying a static local variable)
  - modifying a mutable argument passed by reference
  - performing I/O or calling other side-effect functions

AI전문가과정 - Python Review

List Processing

Methods associated with Lists

`list.append(x)` Add an item to the end of the list. Equivalent to `a[len(a):] = [x]`

`list.extend(iterable)` Extend the list by appending all the items from the iterable. Equivalent to `a[len(a):] = iterable`

`list.insert(i, x)` Insert an item at a given position. The first argument is the index of the element before which to insert

`list.remove(x)` Remove the first item from the list whose value is equal to x. It raises a ValueError if there is no such item

`list.pop([i])` Remove the item at the given position in the list, and return it. If no index is specified, `a.pop()` removes and returns the last item in the list

`list.clear()` Remove all items from the list. Equivalent to `del a[:]`

`list.index(x[, start[, end]])` Return zero-based index in the list of the first item whose value is equal to x. Raises a ValueError if there is no such item

`list.count(x)` Return the number of times x appears in the list

`list.sort(key=None, reverse=False)` Sort the items of the list in place.

`list.reverse()` Reverse the elements of the list in place.

`list.copy()` Return a shallow copy of the list. Equivalent to `a[:]`

When Cloning is Necessary

- Avoid mutating a list over which one is iterating!

- an internal counter for a `for` loop is incremented at the end of each iteration
- the loop ends when the counter reaches the current length of the list

```
def removeDups(L1, L2):
    """
        Assumes that L1 and L2 are lists.
        Removes any element from L1 that also occurs in L2
    """
    for e1 in L1:
        if e1 in L2:
            L1.remove(e1)

L1 = [1, 2, 3, 4]
L2 = [1, 2, 5, 6]
removeDups(L1, L2)
print('L1 = ', L1) # prints L1 = [2, 3, 4]
```

Run Code | Visualize

When Cloning is Necessary

- Iterate over a clone of the original list!
- All slice operations return a new list containing the requested elements
- `L1[:]` is evaluated just once before the first iteration!

```
def removeDups(L1, L2):
    """
       Assumes that L1 and L2 are lists.
       Removes any element from L1 that also occurs in L2
    """
    for e1 in L1[:]:
        if e1 in L2:
            L1.remove(e1)

L1 = [1, 2, 3, 4]
L2 = [1, 2, 5, 6]
removeDups(L1, L2)
print('L1 = ', L1) # prints L1 = [3, 4]
```

Run Code Visualize

Processing a List

- Avoid mutating a list over which one is iterating!
- It is sometimes tempting to change a list while you are looping over it
- however, it is often simpler and safer to create a new list instead

```
import math
raw_data = [56.2, float('NaN'), 51.7, 55.3, 52.5, float('NaN'), 47.8]
filtered_data = []
for value in raw_data:
    if not math.isnan(value):
        filtered_data.append(value)

print(filtered_data)
```

Run Code Visualize

List Comprehension

- a concise way to appy an operation to each element in a sequence
- it creates a new list in which each element is the result of applyting the given operation to each element in the sequence

```
L = [1, 2, 3, 4, 5]
L2 = [x**2 for x in L]
print(L2)  # prints [1, 4, 9, 16, 25]

squares = [x**2 for x in range(10)]
# is equivalent to
squares = list(map(lambda x: x**2, range(10)))
print(squares)
```

Run Code Visualize

`map(function, iterable, ...)`

- Return an iterator that applies `function` to every item of iterable, yielding the results

List Comprehension

```
>>> [(x, y) for x in [1, 2, 3] for y in [3, 1, 4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

is equivalent to:

```
>>> combs = []
>>> for x in [1, 2, 3]:
        for y in [3, 1, 4]:
            if x != y:
                combs.append((x, y))

>>> combs
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

- Note that the order of the `for` and `if` statements is the same in both snippets

List Comprehension

```
>>> vec = [-4, -2, 0, 2, 4]
>>> # apply an operation (e.g., a function) to all the elements
>>> [abs(x) for x in vec]
[4, 2, 0, 2, 4]
>>>
>>> vec = [[1,2,3], [4,5,6], [7,8,9]]
>>> [num for elem in vec for num in elem]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>
>>> from math import pi
>>> [str(round(pi, i)) for i in range(1, 6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

`round(number[, ndigits])`

- Return number rounded to ndigits precision after the decimal point
- If `ndigits` is omitted or is `None`, it returns the nearest integer to its input

Nested List Comprehension

```
matrix = [
    [1, 2, 3, 4],
    [5, 6, 7, 8],
    [9, 10, 11, 12],
]

transposed = [[row[i] for row in matrix] for i in range(4)]
print(transposed)
# prints [[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

- the nested listcomp is evaluated in the context of the `for` that follows it
- the nested listcomp above is equivalent to:

```
transposed = []
for i in range(4):
    transposed.append([row[i] for row in matrix])

print(transposed)
```

built-in function `zip`

```
>>> x = [1, 2, 3]
>>> y = [4, 5, 6]
>>> zipped = zip(x, y)
>>> list(zipped)
[(1, 4), (2, 5), (3, 6)]
```

`zip(*iterables)`

- Returns an iterator of tuples, where the `i`-th tuple contains the `i`-th element from each of the argument sequences or iterables

zip and unpacking

```
>>> x = [1, 2, 3]
>>> y = [4, 5, 6]
>>> zipped = zip(x, y)
>>> list(zipped)          # iterator 'zipped' is exhausted
[(1, 4), (2, 5), (3, 6)]
>>> zipped = zip(x, y)    # get the iterator again
>>> print(*zipped)        # unpacking zipped
(1, 4) (2, 5) (3, 6)
>>> zipped = zip(x, y)    # get the iterator again
>>> zzipped = zip(*zipped)
>>> list(zzipped)
[(1, 2, 3), (4, 5, 6)]
>>>
>>>
>>> x2, y2 = zip(*zip(x, y))  # unpacking and zip
>>> x == list(x2) and y == list(y2)
True
```

Transpose a matrix with unpack and `zip`

```
>>> matrix = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12],]
>>> print(*matrix)
[1, 2, 3, 4] [5, 6, 7, 8] [9, 10, 11, 12]
>>> a, b, c = matrix  # unpacking the matrix list
>>> a
[1, 2, 3, 4]
>>> b
[5, 6, 7, 8]
>>> c
[9, 10, 11, 12]
>>> list(zip(a, b, c))
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
>>>
>>> a, *b = matrix
>>> a
[1, 2, 3, 4]
>>> b
[[5, 6, 7, 8], [9, 10, 11, 12]]
```

Transpose a matrix with unpack and `zip`

```
>>> matrix = [
    [1, 2, 3, 4],
    [5, 6, 7, 8],
    [9, 10, 11, 12],
]
>>> list(zip(*matrix))  # unpacking and zip
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
>>>
>>> result=[list(e) for e in zip(*matrix)]  # list comprehension
>>> result
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

Higher-order Programming with List and Function

- Functions are first-class objects like objects of any other type, e.g., `int`, `float` or `list`
- Higher-order programming
  - an individual function operates on functions.
  - using functions as arguments of other functions

```
def apply2Each(L, f): # defining a high-order function
    """Assumes L is a list, f a function
       Mutates L by replacing each element, e, of L by f(e)"""
    for i in range(len(L)):
        L[i] = f(L[i])

L = [1, -2, 3.33]
apply2Each(L, abs)
print('L = ', L)   # prints L = [1, 2, 3.33]
apply2Each(L, int)
print('L = ', L)   # prints L = [1, 2, 3]
```

Run Code | Visualize

Built-in Higher-order Functions

`map(function, iterable, ...)`

- applies `function` to every item of iterable, returning a sequence (iterator)

```
squares = list(map(lambda x: x**2, range(10)))
print(squares)

for i in map(round, [1.3, 2.6, 3.2]):
   print (i)

L1 = [1, 28, 36]
L2 = [2, 57, 9]
mins = list(map(min, L1, L2))
print(mins)

L = []
for i in map(lambda x, y: x**y, [1, 2, 3, 4], [3, 2, 1, 0]):
   L.append(i)
print(L)
```

Run Code Visualize

Built-in Higher-order Functions

- Built-in function: `filter(function, iterable)`
  - Construct an iterator from those elements of `iterable` for which `function` returns `True`

```
source_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

filtered_list = list(filter(lambda x: (x % 2 == 0), source_list))

print(filtered_list)   # prints [2, 4, 6, 8, 10]
```

Run Code Visualize

Built-in Higher-order Functions

- Built-in function: `sorted(iterable, *, key=None, reverse=False)`
  - Return a new sorted `list` from the items in an iterable
  - Has two optional arguments which must be specified as keyword arguments
  - key specifies a function of one argument that is used to extract a comparison key from each element in iterable
    - The default value is `None` (compare the elements directly)
  - reverse is a `boolean` value. If set to `True`, then the list elements are sorted as if each comparison were reversed

```
result = sorted([5, 2, 3, 1, 4])
print(result)  # prints [1, 2, 3, 4, 5]

result = sorted("This is a test string from Andrew".split(), key=str.lower)
print(result) # prints ['a', 'Andrew', 'from', 'is', 'string', 'test', 'This']
```

Run Code | Visualize

Built-in Higher-order Functions

```
student_tuples = [
    ('john', 'A', 15),
    ('jane', 'B', 12),
    ('dave', 'B', 10),
]
result = sorted(student_tuples, key=lambda student: student[2])   # sort by age
print(result)
# prints [('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
```

Run Code Visualize

Other Higher-order Functions

```
from functools import reduce # import reduce from the module 'functools'

result = reduce(lambda x, y: x+y, [1, 2, 3, 4, 5])
# calculates ((((1+2)+3)+4)+5)
print(result)   # prints 15
```

Run Code | Visualize

- functools.`reduce(function, iterable[, initializer])`
  - Apply `function` of two arguments cumulatively to the items of sequence, from left to right, so as to reduce the sequence to a single value

Comparison of Sequence Types

Type	Type of elements	Examples of literals	Mutable
`str`	characters	'', 'a', 'abc'	No
`tuple`	any type	(), (3,), ('abc', 4)	No
`range`	integers	rang(10), range(1, 10, 2)	No
`list`	any type	[], [3], ['abc', 4.0]	Yes

Common Operations on Sequence Types

- `seq[i]`
- `len(seq)`
- `seq1 + seq2` (not available for ranges)
- `n * seq or seq * n` (not available for ranges)
- `seq[start:end]`
- `e in seq`
- `e not in seq`
- `for e in seq:`

AI전문가과정 - Python Review

Set

Sets

- A set is an unordered collection with no duplicate elements
  - compound data type, container
- `set` is mutable
- Basic uses include membership testing and eliminating duplicate entries
- union, intersection, difference, and symmetric difference
- Curly braces or the `set()` function can be used to create sets
- Note: to create an empty set, you have to use `set()`, not `{}`;
  - the latter creates an empty `dictionary`

Set Construction and Membership Testing

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> print(basket)                      # show that duplicates have been removed
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket                 # fast membership testing
True
>>> 'crabgrass' in basket
False
```

Set Operations

```
>>> # Demonstrate set operations on unique letters from two words
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                      # unique letters in a (duplicates removed)
{'a', 'r', 'b', 'c', 'd'}
>>> b                      # unique letters in b (duplicates removed)
{'a', 'l', 'm', 'c', 'z'}
>>> a - b                  # letters in a but not in b (difference)
{'r', 'd', 'b'}
>>> a | b                  # letters in a or b or both (union)
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                  # letters in both a and b (intersection)
{'a', 'c'}
>>> a ^ b                  # letters in a or b but not both (symmetric difference)
{'r', 'd', 'b', 'm', 'z', 'l'}
```

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

Set Operations

```
len(s)
x in s
x not in s
isdisjoint(other)

issubset(other)
set <= other

set < other

issuperset(other)
set >= other

set > other

union(*others)
set | other | ...

intersection(*others)
set & other & ...

difference(*others)
set - other - ...

symmetric_difference(other)
set ^ other
```

```
update(*others)
set |= other | ...

intersection_update(*others)
set &= other & ...

difference_update(*others)
set -= other | ...

symmetric_difference_update(other)
set ^= other

add(elem)
remove(elem)
discard(elem) # no exception
pop() # Remove and return an arbitrary element
clear()
```

Set Comprehension

```
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
{'r', 'd'}
```

the set comprehension above is equivalent to:

```
a = set();
for x in 'abracadabra':
    if x not in 'abc':
        a.add(x)

print(a)
```

Run Code Visualize

AI전문가과정 - Python Review

Dictionary

Dictionaries

- Objects of type `dict`
- `Dictionary`: a `set` of key-value pairs
  - dictionaries are indexed by keys, which can be any immutable type
    - strings and numbers can always be keys, but not lists
  - the keys are unique (within one dictionary)
  - `{}`: creates an empty dictionary

```
monthNumbers = {'Jan':1, 'Feb':2, 'Mar':3, 'Apr':4, 'May':5,
                1:'Jan', 2:'Feb', 3:'Mar', 4:'Apr', 5:'May'}
print('The thrid month is ' + monthNumbers[3])
dist = monthNumbers['Apr'] - monthNumbers['Jan']
print('Apr and Jan are', dist, 'months apart')
```

Run Code | Visualize

- Entries in a `dict` are
  - ~~unordered~~ insertion ordered and cannot be accessed with an index
    - `monthNumber[1]` refers to the entry with the key `1` (not the second entry)

Dictionaries are mutable!

```
monthNumbers = {'Jan':1, 'Feb':2, 'Mar':3, 'Apr':4, 'May':5,
                1:'Jan', 2:'Feb', 3:'Mar', 4:'Apr', 5:'May'}
monthNumbers['June'] = 6  # add an entry
monthNumbers[6] = 'June'  # add an entry
monthNumbers['May'] = 'V' # update an entry
```

Run Code | Visualize

- Associative retrieval of a value by a key is quite fast
  - the lookup can be done in time that is nearly independent of the size of the dictionary
    - constant-time `O(1)` lookup on average
  - hashing technique

Dictionary Construction

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> {x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
>>> dict(sape = 4139, guido = 4127, jack = 4098) # using keyword arguments
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

Dictionary Operations

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'jack': 4098, 'sape': 4139, 'guido': 4127}
>>> tel['jack']
4098
>>> del tel['sape']   # delete an entry
>>> tel['irv'] = 4127
>>> tel
{'jack': 4098, 'guido': 4127, 'irv': 4127}
>>> list(tel)      # returns a list of all the keys
['jack', 'guido', 'irv']
>>> sorted(tel)    # returns a list of all the keys after sorting them
['guido', 'irv', 'jack']
>>> 'guido' in tel # To check whether a single key is in the dictionary
True
>>> 'jack' not in tel
False
```

Arbitrary Argument Lists

Looping Techniques

```
knights = {'gallahad': 'the pure', 'robin': 'the brave'}
for k, v in knights.items():
    print(k, v)  # prints key/value pairs

for k in knights:
    print(k)  # prints only the keys

for i, v in enumerate(['tic', 'tac', 'toe']):
    print(i, v)  # prints index and value

for i, v in enumerate(knights):
    print(i, v)  # prints keys with index
```

Run Code Visualize

Comparing Sequences and Other Types

- Sequence objects typically may be compared to other objects with the same sequence type

```
(1, 2, 3)              < (1, 2, 4)
[1, 2, 3]              < [1, 2, 4]
'ABC' < 'C' < 'Pascal' < 'Python'
(1, 2, 3, 4)           < (1, 2, 4)
(1, 2)                 < (1, 2, -1)
(1, 2, 3)             == (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab'))   < (1, 2, ('abc', 'a'), 4)
```

Comparing Sequences and Other Types

- The comparison uses lexicographical ordering:
  - first, the first two items are compared, and
  - if they differ, this determines the outcome of the comparison
  - if they are equal, the next two items are compared, and so on, until either sequence is exhausted
- If all items of two sequences compare equal, the sequences are considered equal
- If two items to be compared are themselves sequences of the same type, the lexicographical comparison is carried out recursively
- If one sequence is an initial sub-sequence of the other, the shorter sequence is the smaller (lesser) than the longer

Wrap-up

- All data in a Python program is represented by **objects** or by relations between **objects**.
  - Every object has an identity, a type and a value.
- Object
- String
- Tuple
- List
- List Processing
- Set
- Dictionary

AI전문가과정 - Python Review

Q&A

Acknowledgement

- The Python Tutorial, https://docs.python.org/3/tutorial/index.html
- Lecture Notes, Professor Hyungjoo Kim
- Starting out with Python, Professor Tony Gaddis and Pearson Education, Ltd.
- Introduction to Computation and Programming Using Python, John V. Guttag
- The Python Language Reference Manual, Chris Sheridan, Lulu Press, Inc, 2016