PYTHON – COLLECTION OBJECT TYPES

Lists

- The Python list object is the most general sequence provided by the language.
- Lists are positionally ordered collections of arbitrarily typed objects, and they have no fixed size.
- They are also mutable—unlike strings, lists can be modified in-place by assignment to offsets as well as a variety of list method calls.

Lists

- Because they are sequences, lists support all the sequence operations seen for strings
- Results are usually lists instead of strings

```
>>> L = [123, 'spam', 1.23]          # A list of three different-type objects
>>> len(L)                           # Number of items in the list
3


>>> L[0]                             # Indexing by position
123

>>> L[:-1]                           # Slicing a list returns a new list
[123, 'spam']

>>> L + [4, 5, 6]                    # Concatenation makes a new list too
[123, 'spam', 1.23, 4, 5, 6]

 >>> L                               # We're not changing the original list
 [123, 'spam', 1.23]
```

List methods

append, count, extend, index, insert, pop, remove, reverse, sort

```
>>> L.append('NI')                    # Growing: add object at end of list
>>> L
[123, 'spam', 1.23, 'NI']

>>> L.pop(2)                          # Shrinking: delete an item in the middle
1.23

>>> L                                 # "del L[2]" deletes from a list too
[123, 'spam', 'NI']
```

Lists are mutable:

```
>>> M = ['bb', 'aa', 'cc']
>>> M.sort()
>>> M
['aa', 'bb', 'cc']
>>> M.reverse()
>>> M
['cc', 'bb', 'aa']
```

Access

Python still doesn't allow us to reference items that are not present.

```
>>> L
[123, 'spam', 'NI']

>>> L[99]
...error text omitted...
IndexError: list index out of range
```

```
>>> L[99] = 1
...error text omitted...
IndexError: list assignment index out of range
```

Nested List

```
>>> M = [[1, 2, 3],                    # A 3 x 3 matrix, as nested lists
        [4, 5, 6],                     # Code can span lines if bracketed
        [7, 8, 9]]
>>> M
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]

>>> M[1]                               # Get row 2
[4, 5, 6]

>>> M[1][2]                            # Get row 2, then get item 3 within the row
6
```

```
>>> m1=[1,2,3,4]
>>> m2=[1,2,3,m1]
>>> m3=[1,2,3,m2]
>>> m3
[1, 2, 3, [1, 2, 3, [1, 2, 3, 4]]]
>>> m3[3][3][1]
2
>>>
```

List Comprehension Expression

Comprehensions: Python includes a more advanced operation known as a *list comprehension expression*, which turns out to be a powerful way to process structures like our matrix.

```
>>> col2 = [row[1] for row in M]          # Collect the items in column 2
>>> col2
[2, 5, 8]

>>> M                                     # The matrix is unchanged
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

```
>>> diag = [M[i][i] for i in [0, 1, 2]]   # Collect a diagonal from matrix
>>> diag
[1, 5, 9]

>>> doubles = [c * 2 for c in 'spam']     # Repeat characters in a string
>>> doubles
['ss', 'pp', 'aa', 'mm']
```

List Comprehension Expression

lists, sets, and dictionaries can all be built with comprehensions in 3.0

```
>>> [ord(x) for x in 'spaam']                # List of character ordinals
[115, 112, 97, 97, 109]
>>> {ord(x) for x in 'spaam'}                # Sets remove duplicates
{112, 97, 115, 109}
>>> {x: ord(x) for x in 'spaam'}             # Dictionary keys are unique
{'a': 97, 'p': 112, 's': 115, 'm': 109}
```

Dictionaries

- Not sequences, they are known as *mappings*.
- Collections of other objects stored by key.
- Don’t maintain any reliable order; they simply map keys to associated values.
- Dictionaries are mutable: they may be changed in-place and can grow and shrink on demand, like lists.

```
>>> D = {'food': 'Spam', 'quantity': 4, 'color': 'pink'}

>>> D['food']                  # Fetch value of key 'food'
'Spam'

>>> D['quantity'] += 1         # Add 1 to 'quantity' value
>>> D
{'food': 'Spam', 'color': 'pink', 'quantity': 5}
```

Dictionaries

Another example

```
>>> D = {}
>>> D['name'] = 'Bob'       # Create keys by assignment
>>> D['job']  = 'dev'
>>> D['age']  = 40

>>> D
{'age': 40, 'job': 'dev', 'name': 'Bob'}

>>> print(D['name'])
Bob
```

Dictionary manipulation

```
>>> rec = {'name': {'first': 'Bob', 'last': 'Smith'},
           'job':  ['dev', 'mgr'],
           'age':  40.5}
```

Dictionary manipulation

```
>>> rec['name']                          # 'name' is a nested dictionary
{'last': 'Smith', 'first': 'Bob'}

>>> rec['name']['last']                  # Index the nested dictionary
'Smith'

>>> rec['job']                           # 'job' is a nested list
['dev', 'mgr']
>>> rec['job'][-1]                       # Index the nested list
'mgr'

>>> rec['job'].append('janitor')         # Expand Bob's job description in-place
>>> rec
{'age': 40.5, 'job': ['dev', 'mgr', 'janitor'], 'name': {'last': 'Smith',
'first': 'Bob'}}
```

- Notice how this hybrid object can be expanded at will..

Python garbage collection

- It cleans up unused memory as your program runs and frees you from having to manage such details in your code.
- In Python, the space is reclaimed immediately, as soon as the last reference to an object is removed.

```
>>> rec
{'age': 40.5, 'job': ['dev', 'mgr', 'janitor'], 'name': {'last': 'Smith',
'first': 'Bob'}}

>>> rec = 0                                  # Now the object's space is reclaimed
```

Dictionary to List

```
>>> D = {'a': 1, 'b': 2, 'c': 3}
>>> D
{'a': 1, 'c': 3, 'b': 2}

>>> Ks = list(D.keys())                  # Unordered keys list
>>> Ks                                   # A list in 2.6, "view" in 3.0: use list()
['a', 'c', 'b']

>>> Ks.sort()                            # Sorted keys list
>>> Ks
['a', 'b', 'c']

>>> for key in Ks:                       # Iterate though sorted keys
        print(key, '=>', D[key])         # <== press Enter twice here

a => 1
b => 2
c => 3
```

- Note that the sorted operation is smart enough to figure out that D is a dictionary and not a list..

```
>>> D
{'a': 1, 'c': 3, 'b': 2}

>>> for key in sorted(D):
        print(key, '=>', D[key])

a => 1
b => 2
c => 3
```

Iterations and Conditions

```
>>> for c in 'spam':
        print(c.upper())

S
P
A
M
```

```
>>> x = 4
>>> while x > 0:
        print('spam!' * x)
        x -= 1

spam!spam!spam!spam!
spam!spam!spam!
spam!spam!
spam!
```

Dictionary -- Missing keys

```
>>> D
{'a': 1, 'c': 3, 'b': 2}

>>> D['e'] = 99                    # Assigning new keys grows dictionaries
>>> D
{'a': 1, 'c': 3, 'b': 2, 'e': 99}

>>> D['f']                         # Referencing a nonexistent key is an error
...error text omitted...
KeyError: 'f'
```

- Test for missing keys:

```
>>> 'f' in D
False

>>> if not 'f' in D:
        print('missing')

missing
```

Dictionary -- Missing keys

- Use a built in method to check for keys:

```
>>> value = D.get('x', 0)                    # Index but with a default
>>> value
0
>>> value = D['x'] if 'x' in D else 0        # if/else expression form
>>> value
0
```

Tuples

- An immutable list.
- Tuples are sequences, like lists, but they are immutable, like strings.
- They support arbitrary types, arbitrary nesting, and the usual sequence operations.

```
>>> T = (1, 2, 3, 4)                    # A 4-item tuple
>>> len(T)                              # Length
4

>> T + (5, 6)                           # Concatenation
(1, 2, 3, 4, 5, 6)

>>> T[0]                                # Indexing, slicing, and more
1
```

Files

- Python sees files as objects.
- To create a file object, you call the built-in open function, passing in an external filename and a processing mode as strings.

```
>>> f = open('data.txt', 'w')          # Make a new file in output mode
>>> f.write('Hello\n')                 # Write strings of bytes to it
6
>>> f.write('world\n')                 # Returns number of bytes written in Python 3.0
6
>>> f.close()                          # Close to flush output buffers to disk

>>> f = open('data.txt')               # 'r' is the default processing mode
>>> text = f.read()                    # Read entire file into a string
>>> text
'Hello\nworld\n'

>>> print(text)                        # print interprets control characters
Hello
world

>>> text.split()                       # File content is always a string
['Hello', 'world']
```

File methods

```
>>> dir(f)
[ ...many names omitted...
'buffer', 'close', 'closed', 'encoding', 'errors', 'fileno', 'flush', 'isatty',
'line_buffering', 'mode', 'name', 'newlines', 'read', 'readable', 'readline',
'readlines', 'seek', 'seekable', 'tell', 'truncate', 'writable', 'write',
'writelines']
```

Python distinguishes between text and binary files..

```
>>> data = open('data.bin', 'rb').read()        # Open binary file
>>> data                                        # bytes string holds binary data
b'\x00\x00\x00\x07spam\x00\x08'
>>> data[4:8]
b'spam'
```

Other File-Like Tools

- Python comes with additional file-like tools:
  - pipes, FIFOs, sockets, keyed-access files, descriptor-based files, relational and object-oriented database interfaces, and more.

Sets

- An unordered collections of unique and immutable objects.

```
>>> X = set('spam')                    # Make a set out of a sequence in 2.6 and 3.0
>>> Y = {'h', 'a', 'm'}                # Make a set with new 3.0 set literals
>>> X, Y
({'a', 'p', 's', 'm'}, {'a', 'h', 'm'})

>>> X & Y                              # Intersection
{'a', 'm'}

>>> X | Y                              # Union
{'a', 'p', 's', 'h', 'm'}

>>> X - Y                              # Difference
{'p', 's'}

>>> {x ** 2 for x in [1, 2, 3, 4]}     # Set comprehensions in 3.0
{16, 1, 4, 9}
```

Checking object types

```
# In Python 2.6:

>>> type(L)                          # Types: type of L is list type object
<type 'list'>
>>> type(type(L))                    # Even types are objects
<type 'type'>

# In Python 3.0:

>>> type(L)                          # 3.0: types are classes, and vice versa
<class 'list'>

>>> type(type(L))
<class 'type'>
```

Checking object types

```
>>> if type(L) == type([]):                # Type testing, if you must...
        print('yes')

yes
>>> if type(L) == list:                    # Using the type name
        print('yes')

yes
>>> if isinstance(L, list):                # Object-oriented tests
        print('yes')

yes
```

User-Defined Classes

- Classes define new types of objects that extend the core set

```
>>> class Worker:
        def __init__(self, name, pay):          # Initialize when created
            self.name = name                    # self is the new object
            self.pay  = pay
        def lastName(self):
            return self.name.split()[-1]        # Split string on blanks
        def giveRaise(self, percent):
            self.pay *= (1.0 + percent)         # Update pay in-place
```

```
>>> bob = Worker('Bob Smith', 50000)        # Make two instances
>>> sue = Worker('Sue Jones', 60000)        # Each has name and pay attrs
>>> bob.lastName()                          # Call method: bob is self
'Smith'
>>> sue.lastName()                          # sue is the self subject
'Jones'
>>> sue.giveRaise(.10)                      # Updates sue's pay
>>> sue.pay
66000.0
```

dir(__builtins__)

Built-in Functions				
abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	

```
>>> help(max)
Help on built-in function max in module builtins:

max(...)
    max(iterable[, key=func]) -> value
    max(a, b, c, ...[, key=func]) -> value

    With a single iterable argument, return its largest item.
    With two or more arguments, return the largest argument.
```

In Class Exercise

Write a program that prompts the user for a string and places the string in a list.

When the user types “end” the program prints out the list – one entry per line – and stops.