Chapter 3 - Collection Data Types

CS 171 - Computer Programming 1
Lanzhou University
杨禸
18919801127
yy@lzu.edu.cn

These slides use many elements provided in the main bibliographic reference for these lectures:

Programming in Python 3
A Complete Introduction to the Python Language,
2nd Edition,
Mark Summerfield

Outline

Outline

Sequence Types

- Support the membership operator `in`;
- Support the size function `len()`;
- Support slices `[]`;
- And are iterable;

Python provides 5:

- `bytearray`, and `bytes`, covered later
- `str`, covered in the previous chapter
- `tuple` and `list`, covered in this chapter

Tuples

- Are ordered sequences of zero or more object references;
- Support the same slicing and striding syntax as strings;
- Are immutable;
- If we want to modify an ordered sequence
  - We should use a list;
- If we already have a tuple but want to modify it
  - We can convert it to a list using `list()`
  - and then apply the changes

Tuples

```
>>> t = tuple()    # creates an empty tuple
>>> t
()

>>> t = tuple((5,2,3,4,5))    # passing (no more than) one
>>> t                         # argument to tuple() is possible
(5, 2, 3, 4, 5)               # NOTE: tuple(5,2,3,4,5) is not possible!

>>> t = ()         # tuples can also be created directly
>>> t
()

>>> t = (5,2,3,4,5)
>>> t
(5, 2, 3, 4, 5)
```

- Sometimes, tuples need be enclosed in parentheses to avoid ambiguity

Tuples

```
>>> t
('venus', -28, 'green', '21', 19.74)
```

```
>>> t[0]
'venus'
>>> t[1]
-28
>>> t[2]
'green'
>>> t[3]
'21'
>>> t[4]
19.74
```

```
>>> t[-1]
19.74
>>> t[-2]
'21'
>>> t[-3]
'green'
>>> t[-4]
-28
>>> t[-5]
'venus'
```

Tuples

- Provide just two methods
  - `t.count(x)`, returns the number of times `x` occurs in tuple `t`
  - `t.index(x)`, returns the index of the leftmost occurrence of `x` in `t`
    - or raises `ValueError` is there is no `x` in `t`

- Can be used with the operators
  - `+`, for concatenation
  - `*`, for replication
  - `[]`, for slicing
- And with
  - `in`, and
  - `not in`, to test for membership

- The augmented assignment operators `+=` and `*=` are also available

Tuples

```
>>> hair = "black", "brown", "blonde", "red"
>>> hair[2]
'blonde'
>>> hair[-3:]        # same as: hair[1:]
('brown', 'blonde', 'red')

>>> new_t = hair[:2], "gray", hair[2:]
>>> new_t
(('black', 'brown'), 'gray', ('blonde', 'red'))
>>> len(new_t)
3      # indeed, the tuple above has 3 elements, two of which are tuples

>>> new_t = hair[:2] + ("gray",) + hair[2:]
                     # notice that "gray" is now inside a tuple
>>> new_t
('black', 'brown', 'gray', 'blonde', 'red')
>>> len(new_t)
5
```

Tuples

```
>>> hair = "black", "brown", "blonde", "red"
>>> eyes = ("brown", "hazel", "amber", "green", "blue", "gray")
>>> colors = (hair, eyes)
           # this is another tuple that has tuples as elements

>>> colors[1][3:-1]
           # first we are accessing the tuple on index 1, i.e., eyes
('green', 'blue')

>>> things = (1, -7.5, ("pea", (5, "Xyz"), "queue"))
           # nesting can have as many levels of depth as necessary
>>> things[2][1][1][2]
           # what would the result be here? why?
```

Named Tuples

- Behave just like plain tuples;
- Have the same performance;
- Add the possibility of to refer items in the tuple by their name

```
>>> Sale = collections.namedtuple("Sale",
              "productid customerid date quantity price")
```

- The first argument is the name of the custom tuple
- The second is a string of space-separated names, one per item in the custom tuple
- Having defined it, we can use `Sale` as any other Python class

Named Tuples

- The first argument is the name of the custom tuple
- The second is a string of space-separated names, one per item in the custom tuple
- Having defined it, we can use `Sale` as any other Python class

```
>>> Sale = collections.namedtuple("Sale",
              "productid customerid date quantity price")

>>> sales = []
>>> sales.append(Sale(432, 921, "2008-09-14", 3, 7.99))
>>> sales.append(Sale(419, 874, "2008-09-15", 1, 18.49))

>>> sales
[Sale(productid=432, customerid=921,
           date='2008-09-14', quantity=3, price=7.99),
 Sale(productid=419, customerid=874,
           date='2008-09-15', quantity=1, price=18.49)]
```

Named Tuples

- We can (still) refer to items in the tuples by their index position

```
>>> Sale = collections.namedtuple("Sale",
              "productid customerid date quantity price")

>>> sales = []
>>> sales.append(Sale(432, 921, "2008-09-14", 3, 7.99))
>>> sales.append(Sale(419, 874, "2008-09-15", 1, 18.49))

>>> sales
[Sale(productid=432, customerid=921,
           date='2008-09-14', quantity=3, price=7.99),
 Sale(productid=419, customerid=874,
           date='2008-09-15', quantity=1, price=18.49)]

>>> sales[0][-1]
7.99
```

Named Tuples

- But now we can also use names, which is much more convenient

```
>>> Sale = collections.namedtuple("Sale",
              "productid customerid date quantity price")

>>> sales = []
>>> sales.append(Sale(432, 921, "2008-09-14", 3, 7.99))
>>> sales.append(Sale(419, 874, "2008-09-15", 1, 18.49))

>>> sales
[Sale(productid=432, customerid=921,
           date='2008-09-14', quantity=3, price=7.99),
 Sale(productid=419, customerid=874,
           date='2008-09-15', quantity=1, price=18.49)]

>>> sales[0].price
7.99
```

Named Tuples

```
>>> Aircraft = collections.namedtuple("Aircraft",
                    "manufacturer model seating")
>>> Seating = collections.namedtuple("Seating", "minimum maximum")
>>> aircraft = Aircraft("Airbus", "A320-200", Seating(100, 220))
>>> aircraft.seating.maximum
220
```

Lists

- Are ordered sequences of zero or more object references
- Support the same slicing and striding syntax as strings and tuples
- Unlike strings and tuples, are mutable
  - We can replace and delete slices of lists
- The `list` data type can be called as a function, `list()`
  - with no arguments, it returns an empty list
  - with a list argument it returns a shallow copy of the argument
  - with any other argument, it attempts to convert it to a list
- Can also be created without using `list()`
  - an empty list is created by `[]`
  - a non-empty list is created by `[_item0_, _item1_ ..., _itemn_]`
- Can hold items of any data type

Lists

```
>>> L = [-17.5, "kilo", 49, "V", ["ram", 5, "echo"], 7]

>>> L[0]
-17.5
>>> L[1]
'kilo'
>>> L[2]
49
>>> L[3]
'V'
>>> L[4]
['ram', 5, 'echo']
>>> L[5]
7
>>> L[6]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

```
>>> L[-1]
7
>>> L[-2]
['ram', 5, 'echo']
>>> L[-3]
'V'
>>> L[-4]
49
>>> L[-5]
'kilo'
>>> L[-6]
-17.5
```

Lists

- Can be nested, iterated over and sliced, as tuples
- Can (also) be used with the operators
  - `+`, for concatenation
  - `*`, for replication
  - `[]`, for slicing
- And with
  - `in`, and
  - `not in`, to test for membership

- The augmented assignment operators `+=` and `*=` are also available

Lists

Operations and Functions

Syntax	Description
`L.append(x)`	Appends item `x` to the end of `list L`
`L.count(x)`	Returns the number of times item `x` occurs in `list L`
`L.extend(m)`	Appends all of iterable `m`'s items to the end of `list L`
`L += m`	does the same thing
`L.index(x, start, end)`	Returns the index position of the leftmost occurrence of item `x` in `list L` (or in the `start:end` slice of `L`); otherwise, raises a `ValueError` exception
`L.insert(i, x)`	Inserts item `x` into `list L` at index position `int i`
`L.pop()`	Returns and removes the rightmost item of `list L`
`L.pop(i)`	Returns and removes the item at index position `int i` in `L`
`L.remove(x)`	Removes the leftmost occurrence of item `x` from `list L`, or raises a `ValueError` exception if `x` is not found
`L.reverse()`	Reverses `list L` in-place
`L.sort(...)`	Sorts `list L` in-place; this method accepts the same key and reverse optional arguments as the built-in `sorted()`

Lists

- We can iterate over the items in a list using
  - `for item in L:`
- If we want to *process* all the items in a list:

```
for i in range(len(L)):
    L[i] = _process_(L[i])
```

- `range()` returns an iterator that provides integers
  - With one argument `n`, `range()` produces `0, 1, ..., n-1`
- To increment all the numbers of a list of integers:

```
for i in range(len(numbers)):
        numbers[i] += 1
```

Lists

- Slicing can be used to obtain the same functionality as methods:

```
>>> woods = ["Cedar", "Yew", "Fir"]
>>> woods += ["Kauri", "Larch"]
     # the same as woods.extend(["Kauri", "Larch"])
>>> woods
['Cedar', 'Yew', 'Fir', 'Kauri', 'Larch']

>>> woods = ["Cedar", "Yew", "Fir", "Spruce"]
>>> woods[2:2] = ["Pine"]
     # the same as woods.insert(2, "Pine")
>>> woods
['Cedar', 'Yew', 'Pine', 'Fir', 'Spruce']
```

Lists

- Individual items can easily be replaced in a list:

```
>>> woods [2] = "Redwood"
>>> woods
['Cedar', 'Yew', 'Redwood']
```

- Entire slices can be replaced by assigning an iterable to a slice

```
>>> woods = ["Cedar", "Yew", "Fir"]
>>> woods[1:3] = ["Spruce"]
>>> woods
['Cedar', 'Spruce']
```

List Comprehensions

- A list comprehension is an expression and a loop
  - with an optional condition
- The loop is used to generate items for the list
- The condition is used to filter in/out the wanted/unwanted items

```
[_item_ for _item_ in _iterable_]  # the same as list(iterable)

[_expression_ for _item_ in _iterable_]

[_expression_ for _item_ in _iterable_ if _condition_]
```

List Comprehensions

For calculating the leap years from 1900 until 1940

```
>>> leaps = [y for y in range(1900, 1940)
               if (y % 4 == 0 and y % 100 != 0) or (y % 400 == 0)]
>>> leaps
[1904, 1908, 1912, 1916, 1920, 1924, 1928, 1932, 1936]
```

Outline

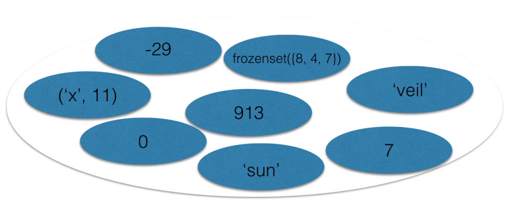
Set Types

- Are a collections data type that supports
  - the membership operator `in`
  - the size function `len()`
  - a `set.isdisjoint()` method
  - comparison
  - bitwise operators
- Python provides two built-in set types:
  - the mutable `set` type
  - the immutable `frozenset`
- When iterated, set types provide their items in an arbitrary order;
- Only *hashable* objects may be added to a set;
  - We will go back to this later

Sets

- Are unordered collections of zero or more object references
- Are mutable
  - So we can easily add/remove items
  - But as they are unordered there is no notion of index position,
    - So, sets cannot be sliced or strided

```
>>> S = {7, "veil", 0, -29, ("x", 11), "sun", frozenset({8, 4, 7}), 913}
```



Sets

- The `set` data type can be called as a function, `set()`
  - with no arguments, it returns an empty set
  - with a set argument it returns a shallow copy of the argument
  - with any other argument, it attempts to convert it to a set
- Nonempty sets can also be created without using the `set()` function
  - but the empty set must be created with `set()`, **not** using braces

- Always contain unique items
  - it is safe, but pointless, to add duplicate items

```
# these are all the same set!
set("apple")
set("aple")
{'e', 'p', 'a', 'l'}
```

Sets

- Provide the usual set operators

```
# Union
>>> set("pecan") | set("pie")
{'a', 'e', 'n', 'p', 'i', 'c'}

# Intersection
>>> set("pecan") & set("pie")
{'p', 'e'}

# Difference
>>> set("pecan") - set("pie")
{'c', 'a', 'n'}

# Symmetric difference
>>> set("pecan") ^ set("pie")
{'a', 'i', 'n', 'c'}
```

Sets - Operations and Functions

Syntax	Description
`s.add(x)`	Adds item `x` to set `s` if it is not already in `s`
`s.clear()`	Removes all the items from set `s`
`s.copy()`	Returns a shallow copy of set `s`*
`s.difference(t)` `s-t`	Returns a new set that has every item that is in set `s` that is not in set `t`*
`s.difference_update(t)` `s -= t`	Removes every item that is in set `t` from set `s`
`s.discard(x)`	Removes item `x` from set `s` if it is in `s`; see also `set.remove()`
`s.intersection(t)` `s&t`	Returns a new set that has each item that is in both set `s` and set `t`*
`s.intersection_update(t)` `s &= t`	Makes set `s` contain the intersection of itself and set `t`
`s.isdisjoint(t)`	Returns `True` if sets `s` and `t` have no items in common*
`s.issubset(t)` `s <= t`	Returns `True` if set `s` is equal to or a subset of set `t` use `s < t` to test whether `s` is a proper subset of `t`*

Sets - Operations and Functions

Syntax	Description
`s.issuperset(t)` `s >= t`	Returns `True` if set `s` is equal to or a superset of set `t`; use `s>t` to test whether `s` is a proper superset of `t`*
`s.pop()`	Returns and removes a random item from set `s`, or raises a `KeyError` exception if `s` is empty
`s.remove(x)`	Removes item `x` from set `s`, or raises a `KeyError` exception if `x` is not in `s`; see also `set.discard()`
`s.symmetric_difference(t)` `s^t`	Returns a new set that has every item that is in set `s` and every item that is in set `t`, but excluding items that are in both sets*
`s.symmetric_difference_update(t)` `s ^= t`	Makes set `s` contain the symmetric difference of itself and set `t`
`s.union(t)` `s|t`	Returns a new set that has all the items in set `s` and all the items in set `t` that are not in set `s`*
`s.update(t)` `s |= t`	Adds every item in set `t` that is not in set `s`, to set `s`

Set Comprehensions

- We can also create sets using set comprehensions
- Which consist of an expression and a loop with an optional condition
- Like list comprehensions, two syntaxes are supported:

```
[_expression_ for _item_ in _iterable_]

[_expression_ for _item_ in _iterable_ if _condition_]
```

- We can use these to achieve a filtering effect:
  - providing the order doesn't matter

```
# what does this piece do?
html = {x for x in files if x.lower().endswith((".htm", ".html"))}
```

Frozen Sets

- Once created, a frozen set cannot be changed
- Can only be created using function `frozenset()`
  - with no arguments, it returns an empty frozen set
  - with one argument it returns a shallow copy of the argument
  - with any other argument, it attempts to convert it to a `frozenset`
- Since they are immutable, from the operations shown earlier for sets, they support only the ones that do not affect/change the `frozenset`
  - These are the ones marked with *

Outline

Mapping Types

- Are collections of key-value items
  - and provide methods for accessing items and their keys and values
- There are unordered mapping types, whose items and provided in arbitrary order
  - the built in `dict`
  - the standard library's `collections.defaultdict`
- And one ordered mapping type
  - `collections.OrderedDict`, which stores items in insertion order
- When it doesn't make a difference, we will refer to both as *dictionaries*
- Only hashable objects may be used as dictionary keys
  - Immutable types as `float`, `frozenset`, `int`, `str`, and `tuple` can be used
  - `dict`, `list`, and `set` cannot
- Values associated with keys can be objects of any type

Dictionaries

- A `dict` is an unordered collection of zero or more key–value pairs
- Are mutable
  - we can add or remove items
  - they have no notion of index position so cannot be sliced or strided
- Can be created using function `dict()`
  - with no arguments, it returns an empty dictionary
  - with one mapping argument it returns a dictionary based on the argument
  - with a sequence argument, if each item in the sequence is itself a sequence of two objects;
    - the first object will be the key and the second the value

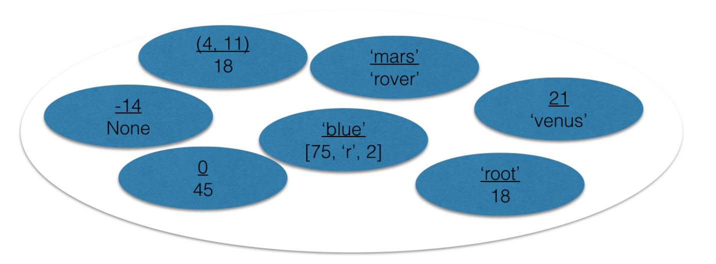
Dictionaries

```
# These are all the same dictionary

d1 = dict({"id": 1948, "name": "Washer", "size": 3})

d2 = dict(id=1948, name="Washer", size=3)

d3 = dict([("id", 1948), ("name", "Washer"), ("size", 3)])

d4 = dict(zip(("id", "name", "size"), (1948, "Washer", 3)))

d5 = {"id": 1948, "name": "Washer", "size": 3}
```

```
>>> d1 = dict({"id": 1948, "name": "Washer", "size": 3})
>>> d1
{'id': 1948, 'name': 'Washer', 'size': 3}

>>> d1["height"] = 178
>>> d1
{'id': 1948, 'name': 'Washer', 'size': 3, 'height': 178}
```

Dictionaries

```
d = {"root": 18, "blue": [75, "R", 2],
     21: "venus", -14: None,
     "mars": "rover",
     (4, 11): 18, 0: 45}
```



Dictionaries - Operations and Functions

Syntax	Description
`d.clear()`	Removes all items from `dict d`
`d.copy()`	Returns a shallow copy of `dict d`
`d.fromkeys(s, v)`	Returns a `dict` whose keys are the items in sequence `s` and whose values are `None` or `v` if `v` is given
`d.get(k)`	Returns key `k`'s associated value, or `None` if `k` isn't in `dict d`
`d.get(k, v)`	Returns key `k`'s associated value, or `v` if `k` isn't in `dict d`
`d.items()`	Returns a view of all the `(key, value)` pairs in `dict d`
`d.keys()`	Returns a view of all the `keys` in `dict d`

Dictionaries - Operations and Functions

Syntax	Description
`d.pop(k)`	Returns key `k`'s associated value and removes the item whose `key` is `k`, or raises a `KeyError` exception if `k` isn't in `d`
`d.pop(k, v)`	Returns key `k`'s associated value and removes the item whose `key` is `k`, or returns `v` if `k` isn't in `dict d`
`d.popitem()`	Returns and removes an arbitrary `(key, value)` pair from `dict d`, or raises a `KeyError` exception if `d` is empty
`d.setdefault(k, v)`	The same as the `dict.get()` method, except that if the key is not in `dict d`, a new item is inserted with the key `k`, and with a `value` of `None` or of `v` if `v` is given
`d.update(a)`	Adds every `(key, value)` pair from `a` that isn't in `dict d` to `d`, and for every `key` that is in both `d` and `a`, replaces the corresponding value in `d` with the one in `a` — `a` can be a dictionary, an iterable of `(key, value)` pairs, or keyword arguments
`d.values()`	Returns a view of all the values in `dict d`

Iterating over Dictionaries

```
#Iterating by (key, value) pairs
for item in d.items():
    print(item[0], item[1])

for key, value in d.items():
    print(key, value)

#Iterating by keys
for key in d:
    print(key)

for key in d.keys():
    print(key)

#Iterating by values
for value in d.values():
    print(value)
```

Dictionary Comprehensions

- Consist of an expression and a loop with an optional condition
  - very similar to a set comprehension

```
{_keyexpression_: _valueexpression_ for _key_,
                        _value_ in _iterable_}
{_keyexpression_: _valueexpression_ for _key_,
                        _value_ in _iterable_ if _condition_}
```

- One example:

```
#what does this piece do?
file_sizes = {name: os.path.getsize(name)
                        for name in os.listdir(".")}
```

Default Dictionaries

- Are dictionaries
  - they have all the operators and methods that dictionaries provide
- However, they handle missing keys differently
- Before, if we used a nonexistent key when accessing a dictionary, a `KeyError` was raised

```
>>> d = dict({1:2, 3:4})
>>> d
{1: 2, 3: 4}
>>> d[1]
2
>>> d[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 4
```

Default Dictionaries

- For defaultdict, this is not the case:

```
>>> l = [(1, 2), (3, 4)]
>>> d = collections.defaultdict(int)
>>> for k, v in l: d[k] = v
...
>>> d
defaultdict(<class 'int'>, {1: 2, 3: 4})
>>> d[1]
2
>>> d[4]
0
>>> d
defaultdict(<class 'int'>, {1: 2, 3: 4, 4: 0})
```

Ordered Dictionaries

- Provide the same functions and methods than unordered `dicts`
- The difference is that ordered dictionaries store their items in the order in which they were inserted

```
>>> tasks = collections.OrderedDict()
>>> tasks[8031] = "Backup"
>>> tasks[4027] = "Scan Email"
>>> tasks[5733] = "Build System"

>>> list(tasks.keys())
[8031, 4027, 5733] # this order is guaranteed!

>>> tasks[8031] = "Daily backup"
>>> list(tasks.keys())
[8031, 4027, 5733] # same order
>>> tasks
OrderedDict([(8031, 'Daily backup'), (4027, 'Scan Email'),
             (5733, 'Build System')])
```

Outline

Copying Collections

- Synce Python uses object references, we need to be really careful!
- When using =, no copying takes place!

```
>>> songs = ["Because", "Boys", "Carol"]
>>> beatles = songs
>>> beatles, songs
    # both variables point to the same list
(['Because', 'Boys', 'Carol'], ['Because', 'Boys', 'Carol'])

>>> beatles[2] = "Cayenne"
    # so, changing the list has impact on both variables
>>> beatles, songs
(['Because', 'Boys', 'Cayenne'], ['Because', 'Boys', 'Cayenne'])
```

Copying Collections

- For sequences, when we take a slice, it is always an independent copy of the items copied

```
>>> songs = ["Because", "Boys", "Carol"]
>>> beatles = songs[:]
>>> beatles[2] = "Cayenne"
>>> beatles, songs
(['Because', 'Boys', 'Cayenne'], ['Because', 'Boys', 'Carol'])
```

- For dictionaries, copying can be achieved with `dict.copy()`
- For sets, copying can be achieved with `set.copy()`
- Alternatively,

```
copy_of_dict_d = dict(d)
copy_of_list_l = list(l)
copy_of_set_s = set(s)
```

Copying Collections

- Still, this does not work to copy with nested structures:

```
>>> x = [53, 68, ["A", "B", "C"]]
>>> y = x[:] # shallow copy
>>> x, y
([53, 68, ['A', 'B', 'C']], [53, 68, ['A', 'B', 'C']])
>>> y[1] = 40
>>> x[2][0] = 'Q'
>>> x, y
([53, 68, ['Q', 'B', 'C']], [53, 40, ['Q', 'B', 'C']])
```

- If this is really what you want, you need deepcopy

```
>>> import copy
>>> x = [53, 68, ["A", "B", "C"]]
>>> y = copy.deepcopy(x)
>>> y[1] = 40
>>> x[2][0] = 'Q'
>>> x, y
([53, 68, ['Q', 'B', 'C']], [53, 40, ['A', 'B', 'C']])
```