

SI 506 Lecture 15 (Halloween edition)

Topics

1. The Dictionary type
2. Creating dictionaries
3. Accessing, adding, modifying, and deleting key-value pairs
4. Dictionary methods

Vocabulary

- **Dictionary.** An associative array or a map, wherein each specified value is associated with or mapped to a defined key that is used to access the value.

Previous

- **Argument.** A value passed to a function or method that corresponds to a parameter defined for the function or method.
- **Boolean.** A type (`bool`) or an expression that evaluates to either `True` or `False`.
- **Built-in Function.** A `function` defined by the Standard Library that is always available for use.
- **Caller.** The initiator of a function call.
- **Conditional Statement.** A statement that determines a computer program's *control flow* or the order in which particular computations are to be executed.
- **Deep copying.** For a given mutable object (e.g., `list`) constructs a new compound object and recursively *copies* into it objects found in the original.
- **Expression.** An accumulation of values, operators, and/or function calls that return a value. `len(<some_list >)` is considered an expression.
- **f-string.** Formatted string literal prefixed with `f` or `F`.
- **File Object.** An object that provides a file-oriented application programming interface (API) to a either a text file, binary file (e.g., image file), or a buffered binary file. File objects include read and write methods for interacting with a file stored locally or remotely.
- **Flow of execution.** The order in which statements in a program are executed. Also referred to as *control flow*.
- **Function.** A defined block of code that performs (ideally) a single task. Functions only run when they are explicitly called. A function can be defined with one or more *parameters* that allow it to accept *arguments* from the caller in order to perform a computation. A function can also be designed to return a computed value. Functions are considered "first-class" objects in the Python eco-system.
- **Immutable.** Object state cannot be modified following creation. Strings are immutable.
- **Iterable.** An object capable of returning its members one at a time. Both strings and lists are examples of an iterable.
- **Iteration.** Repetition of a computational procedure in order to generate a possible sequence of outcomes. Iterating over a `list` using a `for` loop is an example of iteration.
- **Method.** A function defined by and bound to an object. For example the `str` type is provisioned with a number of methods including `str.strip()`.
- **Mutable.** Object state can be modified following creation. Lists are mutable.
- **Nested Loop.** A `for` or `while` loop located within the code block of another loop.

- **Operator.** A [symbol](#) for performing operations on values and variables. The assignment operator (`=`) and arithmetic operators (`+`, `-`, `*`, `/`, `**`, `%`, `//`).
- **Parameter.** A named entity in a function or method definition that specifies an argument that the function or method accepts.
- **Scope.** The part of a script or program in which a variable and the object to which it is assigned is visible and accessible.
- **Sequence.** An ordered set such as `str`, `list`, or `tuple`, the members of which (e.g., characters, elements, items) can be accessed.
- **Shallow copying.** For a given mutable object (e.g., `list`) constructs a new compound object but inserts *references* (rather than copies) into it of objects found in the original. The `list.copy()` returns a shallow copy of the original list.
- **Slice.** A subset of a sequence. A slice is created using the subscript notation `[]` with colons separating numbers when several are given, such as in `variable_name[1:3:5]`. The bracket notation uses slice objects internally.
- **Statement.** An instruction that the Python Interpreter can execute. For example, assigning a variable to a value such as `name = 'arwhyte'` is considered a statement.
- **Truth Value.** In Python any object can be tested for its [truth value](#) using an `if` or `while` condition or when it is used as an operand in a [Boolean operation](#).
- **Tuple.** An ordered sequence that cannot be modified once it is created.
- **Tuple packing.** Assigning items to a tuple.
- **Tuple unpacking.** Assigning tuple items to an equal number of variables in a single assignment. A `list` can also be unpacked.

1.0 The dictionary

Lists and tuples are robust data structures but one downside that they both share is that neither provides explicit hints as regards the meaning of each element or item.

```
film = ['The Wizard of Oz', 1939, 2800000, 26100000, 'The Wicked Witch of the West', 'evil spells']
```

While the first element in the above list is likely comprehensible, discerning the meaning of the other elements may require "insider" knowledge or an accompanying data dictionary. Such a situation is rarely ideal when working with data.

1.1 Syntax

The Python dictionary (`dict`) provides an alternative data structure for both *describing* data and storing values. Python dictionaries (type: `dict`) are considered *associative arrays*, wherein each specified value is associated with or mapped to a defined key that is used to access the value.

The beauty of the dictionary is the ability to identify data values by a label or key, usually rendered as a readable string though not always (integers and tuples are also used as keys). In other words, you can embed meaning into a data structure.



You'll often hear people refer to dictionaries as unordered sets of *key-value pairs*. However, since Python 3.7 dictionary order is now guaranteed to be the insertion order of its key-value pairs.

Dictionaries are defined by enclosing a comma-separated sequence of key-value pairs within curly braces (`{}`).

```
{  
    < key >: < value >,  
    < key >: < value >,  
    ...  
}
```

Each key-value pair is separated by a colon (`:`). Each value specified is referenced by its associated key rather than by its numerical position within the dictionary.

Below is a simple dictionary representation of the classic 1939 film *The Wizard of Oz*:

```
film = {  
    'title': 'The Wizard of Oz',  
    'year_released': 1939,  
    'budget': 2800000,  
    'box_office_receipts': 26100000,  
    'scary_character_name': 'The Wicked Witch of the West',  
    'scary_character_weapon': 'evil spells'  
}  
  
film_type = type(film) # <class 'dict'>
```

2.0 Creating a dictionary

There are several ways to create a dictionary.

2.1 Assign an empty dictionary to a variable

You can create a dictionary by assigning an empty dictionary to a variable and then adding key-value pairs one at a time to the dictionary using the subscript operator (`[]`) (a.k.a bracket notation).



note the nested dictionary `scary_character`.

```
film = {} # empty dict  
film['title'] = 'Psycho'  
film['year_released'] = 1960  
film['budget'] = 806947  
film['box_office_receipts'] = 50000000  
film['scary_character'] = {} # nested dict  
film['scary_character']['name'] = 'Norman Bates'  
film['scary_character']['weapon'] = "chef's knife"
```

2.2 Dictionary literal

You can create a dictionary by defining a dictionary *literal* and assigning keys and values separated by a colon (:).

```
film = {  
    'title': 'Halloween',  
    'year_released': 1978,  
    'budget': 325000,  
    'box_office_receipts': 70000000,  
    'scary_character_name': 'Michael Myers',  
    'scary_character_weapon': "chef's knife"  
}
```

2.3 Built-in dict() function

You can also call the built-in `dict()` function to define a dictionary. You can pass in a sequence of keyword arguments separated by commas or pass in a sequence of tuples.

```
# Pass in keyword arguments (note use of nested dict())  
film = dict(  
    title='Friday the 13th',  
    year_released=1980,  
    budget=5500000,  
    box_office_receipts=59800000,  
    scary_character=dict(  
        name='Jason Vorhees',  
        weapon='machete'  
    )  
)  
  
# Pass in tuples (note used of nested dict())  
film = dict(  
    [  
        ('title', 'Friday the 13th'),  
        ('year_released', 1980),  
        ('budget', 5500000),  
        ('box_office_receipts', 59800000),  
        ('scary_character', dict(  
            [  
                ('name', 'Jason Vorhees'),  
                ('weapon', 'machete')  
            ])  
        )  
    ]  
)
```

3.0 Accessing, adding, modifying, and deleting key-value pairs

3.1 Accessing a dictionary value using subscript notation

A dictionary value is accessed by its associated key.

```
film = {  
    'title': 'A Nightmare on Elm Street',  
    'year_released': 1984,  
    'budget': 1800000,  
    'box_office_receipts': 25500000,  
    'scary_character_name': 'Freddy Krueger',  
    'scary_character_weapon': 'clawed glove'  
}  
  
film_title = film['title']
```

! If you attempt to access a dictionary value with a non-existent key you will trigger a `KeyError` exception.

```
film_name = film['name'] # raises KeyError: 'name'
```

Besides strings, numbers, and booleans dictionaries can reference more complex data structures such as lists, tuples, and other dictionaries. This is often referred to as "nesting".

💡 In the example below the representation of the scary character has been converted to a dictionary of two key-value pairs.

```
film = {  
    'title': 'A Nightmare on Elm Street',  
    'year_released': 1984,  
    'budget': 1800000,  
    'box_office_receipts': 25500000,  
    'scary_character': {  
        'name': 'Freddy Krueger',  
        'weapon': 'clawed glove'  
    }  
}
```

To access nested values you can "chain" the subscript operator `[]`, providing each bracket with appropriate the key, index, or slice as determined by the type of nested object.

```
# Accessing a nested dictionary value  
scary_character_name = film['scary_character']['name']
```

3.2 Add, modify, and delete a key-value pair

You can add a *new* key-value pair to an *existing* dictionary by assigning a new value to the dictionary using the subscript operator (`[]`) and specifying a key.

```
film = {
    'title': 'The Shining',
    'year_released': 1980,
    'budget': 19000000,
    'box_office_receipts': 46200000,
    'scary_character': {
        'name': 'Jack Torrance',
        'weapon': 'axe'
    }
}

# Add key-value pairs
film['director'] = 'Stanley Kubrick'
film['screenplay_authors'] = [film['director'], 'Diane Johnson']
film['stars'] = ['Jack Nicholson', 'Shelley Duvall', 'Scatman Crothers',
'Danny Lloyd']
```

The new key-value pairs are appended to the dictionary's key-value pair sequence.

If you need to *modify* an existing value you can assign a new value by referencing the relevant key:

```
film['box_office_receipts'] = 47000000
```

If you need to delete a key-value pair you can use the built-in `del()` function.

```
del(film['stars'])
```

The above approaches holds true for nested objects as well. Use subscript chaining to reference the relevant key-value pair.

```
# Adding a new key-value pair to a nested dictionary
film['scary_character']['profession'] = 'writer'
```

4.0 Dictionary methods

The Dictionary object is provisioned with several useful [methods](#) of which the following are relevant to today's discussion:

- `dict.get()`

- `dict.keys()`
- `dict.values()`
- `dict.items()`

4.1 `dict.get()` method

You can guard against `KeyError` runtime exceptions by accessing dictionary values using the `dict.get()` method.

```
dict.get(< key >[, < default value >])
```

The `dict.get()` method defines two parameters: a key and an optional default value to return if the passed in key has no associated value. If a default value is not specified, the optional default value defaults to `None`. This behavior prevents `dict.get()` from triggering a `KeyError` if a non-existent key is passed to it.

```
film = {
    'title': 'Scream',
    'year_released': 1996,
    'budget': 15000000,
    'box_office_receipts': 173000000,
    'scary_character_name': 'Ghostface',
    'scary_character_weapon': 'knife'
}

ghostface = film['scary_character_name'] # returns str

# TODO Uncomment
# ghostface = film['scary_character'] # triggers KeyError

ghostface = film.get('scary_character', 'A scary character') # returns
default value

ghostface = film.get('scary_character') # returns None

ghostface = film.get('scary_character_name') # returns str
```

4.2 `dict.keys()` method

You can retrieve all the keys in a dictionary by calling `dict.keys()`. The method returns a `dict_keys` object, an object that provides a *view* or a pointer to the dictionary's keys. While you can loop over a `dict_keys` object you *cannot* modify either the referenced keys or the associated dictionary.

💡 you can create a copy of the `dict_keys` object using the built-in `list()` function. Passing the `dict_keys` object to `list()` will return a list of keys. Doing so simplifies working with the keys.

Below is a simple dictionary representation of [Rotten Tomatoes](#)' ratings of *The Wizard of Oz* (1939) by critics ("tomatometer") and moviegoers ("audience").

```
rating = {
    'title': 'The Wizard of Oz',
    'year_released': 1939,
    'tomatometer_percent_score': .98,
    'tomatometer_avg_rating': 9.50,
    'tomatometer_raters': 160,
    'audience_percent_score': .89,
    'audience_avg_rating': 4.30,
    'audience_raters': 250000
}


# Loop over dict_keys object
for key in rating.keys():
    print(key)

# Convert dict_keys to a list
rating_keys = list(rating.keys())

# Loop over list of keys; print associated values
for key in rating_keys:
    print(rating[key])
```

4.3 `dict.values()` method

You can retrieve all the values in a dictionary by calling `dict.values()`. The method returns a `dict_values` object, an object that provides a *view* or a pointer to the dictionary's values. While you can loop over a `dict_values` object you *cannot* modify either the referenced values or the associated dictionary.

 you can create a copy of the `dict_values` object using the built-in `list()` function. Passing the `dict_values` object to `list()` will return a list of values. Doing so simplifies working with the values.

```
rating = {
    'title': 'Psycho',
    'year_released': 1960,
    'tomatometer_percent_score': .96,
    'tomatometer_avg_rating': 9.22,
    'tomatometer_raters': 101,
    'audience_percent_score': .95,
    'audience_avg_rating': 4.46,
    'audience_raters': 240145
}

# Loop over dict_values object
for value in rating.values():
    print(value)
```



```
# Convert to a list
rating_values = list(rating.values()) # convert to a list

# Print value types
for value in film.values():
    print(type(value))
```

4.4 `dict.items()` method

You can loop over a dictionary's keys *and* values by calling the `dict.items()` method. `dict.items()` returns a `dict_items` object, a list-like object composed of key-value tuples.

```
rating = {
    'title': 'Get Out',
    'year_released': 2017,
    'tomatometer_percent_score': .98,
    'tomatometer_avg_rating': 8.35,
    'tomatometer_raters': 386,
    'audience_percent_score': .86,
    'audience_avg_rating': 4.18,
    'audience_raters': 75450
}

# Looping over a dictionary's items
for key, val in film.items():
    print(f"key: {key} | val: {val}")
```

Call `dict.items()` whenever you need to filter on specific keys in order to access a subset of the dictionary's values. In the following example the the list of films contained in the file `scary_films` is returned as a list of dictionaries after calling the function `read_csv_to_dicts`. Then a nested list is used to access the film names and append them to an accumulator list.

```
filepath = './scary_films.csv'
films = read_csv_to_dicts(filepath) # returns list of dictionaries

# dict.items()
film_names = []
for film in films:
    for key, val in film.items():
        if key == 'title':
            film_names.append(val)

# Alternative dict.keys()
film_names = []
for film in films:
    for key in film.keys():
```

```
if key == 'title':  
    film_names.append(film[key])
```