SI 506 Lecture 16

Topics

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.

Reference

Boomark the following w3schools reference page:

1. w3schools, "Python Dictionary Methods"

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.

```
site = [
    527,
    'Cultural',
    'Kyiv: Saint-Sophia Cathedral and Related Monastic Buildings, Kyiv-
Pechersk Lavra',
    'Designed to rival Hagia Sophia in Constantinople . . . .',
    1990,
    0,
    30.51686,
    50.45258,
    28.52,
    'Europe and North America',
    'Ukraine',
    'UKR'
    ]
```

While a few of the elements in the above list are 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.

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 numercial position within the dictionary.

Below is a simple dictionary representation of Kyiv's 🇺🇦 two UNESCO World Heritage sites: Saint-Sophia Cathedral and Kyiv-Pechersk Lavra (Monastery of the Caves):

```
site = {
    'id_no': 527,
    'category': 'Cultural',
    'name_en': 'Kyiv: Saint-Sophia Cathedral and Related Monastic
Buildings, Kyiv-Pechersk Lavra',
    'short_description_en': 'Designed to rival Hagia Sophia in
Constantinople . . . .',
    'date_inscribed': 1990,
    'endangered': 0,
    'longitude': 30.51686,
    'latitude': 50.45258,
    'area_hectares': 28.52,
    'region_en': 'Europe and North America',
    'states_name_en': 'Ukraine',
    'undp_code': 'UKR'
    }
```

💡 Python dictionary objects are *iterables* with a length or size that can be accessed by calling the built-in function `len()`.

💡 A dictionary can also be provisioned with a *composite* key, that is, a key comprising two or more values. The composite key must be *hashable*, that is, it must never change during its lifetime. Tuples that contain hashable values can be employed as a dictionary key as the following example illustrates

```
# Tuple (< longitude >, < latitude >)
location = {
    (30.51686, 50.45258): 'Kyiv: Saint-Sophia Cathedral and Related
Monastic Buildings, Kyiv-Pechersk Lavra'
}
```

You can check if an object is hashable by passing it to the built-in `hash()` function. Both strings and tuples possess a hash value; lists and dictionaries do not.

```
# Tuple is hashable
hash_value = hash((30.51686, 50.45258)) # -4047005574466108915

# List is not hashable
hash_value = hash([30.51686, 50.45258]) # TypeError: unhashable type:
'list'
```

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 (`[]`).

💡 note the nested dictionary assigned to the key `geo_coordinates`.

```
site = {}
site['id_no'] = 1330
site['category'] = 'Cultural'
site['name_en'] = 'Residence of Bukovinian and Dalmatian Metropolitans'
site['short_description_en'] = '[R]epresents a masterful synergy of architectural styles built by Czech architect Josef Hlavka from 1864 to 1882. . . .'
site['date_inscribed'] = 2011
site['endangered'] = 0
site['geo_coordinates'] = {} # nested dict
site['geo_coordinates']['longitude'] = 25.9247222222
site['geo_coordinates']['latitude'] = 48.2966666667
site['area_hectares'] = 8.0
site['region_en'] = 'Europe and North America'
site['states_name_en'] = 'Ukraine'
site['undp_code'] = 'UKR'
```

2.2 Dictionary literal

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

```
site = {
    'id_no': 865,
    'category': 'Cultural',
    'name_en': "L'viv – the Ensemble of the Historic Centre",
    'short_description_en': "The city of L'viv, founded in the late Middle Ages, was a flourishing administrative, religious and commercial centre for several centuries. The medieval urban topography has been preserved virtually intact . . . .",
    'date_inscribed': 1998,
    'endangered': 0,
    'geo_coordinates': {
        'longitude': 24.03198,
        'latitude': 49.84163
        },
    'area_hectares': 120.0,
    'region_en': 'Europe and North America',
    'states_name_en': 'Ukraine',
    'undp_code': 'UKR',
    }
```

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())
site = dict(
    id_no=1411,
    category='Cultural',
    name_en='Ancient City of Tauric Chersonese and its Chora',
    short_description_en='The site features the remains of a city founded
by Dorian Greeks in the 5th century BC on the northern shores of the Black
Sea. . . .',
    date_inscribed=2013,
    endangered=0,
    geo_coordinates=dict(longitude=33.4913888889, latitude=44.6108333333),
    area_hectares=259.3752,
    region_en='Europe and North America',
    states_name_en='Ukraine',
    undp_code='UKR'
    )

# Pass in tuples (note used of nested dict())
site = dict(
    [
        ('id_no', 1411),
        ('category', 'Cultural'),
        ('name_en', 'Ancient City of Tauric Chersonese and its Chora'),
        ('short_description_en', 'The site features the remains of a city
founded by Dorian Greeks in the 5th century BC on the northern shores of
the Black Sea. . . .'),
        ('date_inscribed', 2013),
        ('endangered', 0),
        ('geo_coordinates', dict([('longitude', 33.4913888889),
('latitude', 44.6108333333)])),
        ('area_hectares', 259.3752),
        ('region_en', 'Europe and North America'),
        ('states_name_en', 'Ukraine'),
        ('undp_code', 'UKR')
        ]
    )
```

3.0 Working with dictionaries

Dictionaries, like lists, are mutable and capable of modification. Utilize the subscript operator `[]` and a key value to interact with a dictionary and its individual key-value pairs.

3.1 Accessing a value

A dictionary value is accessed by its associated key using the subscript operator.

```
site = {
        'id_no': 527,
        'category': 'Cultural',
        'name_en': 'Kyiv: Saint-Sophia Cathedral and Related Monastic
Buildings, Kyiv-Pechersk Lavra',
        'short_description_en': 'Designed to rival Hagia Sophia in
Constantinople . . . .',
        'date_inscribed': 1990,
        'endangered': 0,
        'geo_coordinates': {
            'longitude': 24.03198,
            'latitude': 49.84163
            },
        'area_hectares': 28.52,
        'region_en': 'Europe and North America',
        'states_name_en': 'Ukraine',
        'undp_code': 'UKR'
        }

    # Accessing a value
    site_name = site['name_en']
```

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

```
site_name = site['name'] # raises KeyError: 'name'
```

3.2 Accessing a "nested" value

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".

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

```
# Accessing a nested dictionary value
site_latitude = site['geo_coordinates']['latitude']
```

3.3 Add 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.

```
# Add key-value pair
site['subregion_en'] = 'Eastern Europe'
```

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

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

```
# Add nested key-value pairs
  site['street_address'] = {}
  site['street_address']['Saint-Sophia Cathedral'] = 'Volodymyrska St, 24, Kyiv, Ukraine, 01001'
  site['street_address']['Kyiv Pechersk Lavra'] = 'Lavrska St, 15, Kyiv, Ukraine, 01015'
```

3.4 Modify an existing value

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

```
# Modify existing key-value pair
site['endangered'] = 1
```

3.5 Delete a key-value pair

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

```
# Delete key-value pair
del(site['undp_code'])
```

4.0 Select 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.

```
site_type = site.get('category') # returns str

# TODO Uncomment
# site_type = site['type'] # triggers KeyError

site_type = site.get('type') # returns None

site_type = site.get('type', 'Undefined') # returns default value
```

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.

The following Ukraine 🇺🇦 dictionary comprising basic country data compiled by the United Nations illustrates basic use of the `dict.keys()` method.

```python
country = {
    'name': 'Ukraine',
    'region': 'Eastern Europe',
    'population': 43467000,
    'urban_population_pct': 69.5,
    'surface_area_km2': 603500,
    'capital_city': 'Kyiv',
    'un_membership_date': '1945-10-24'
}

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

# Convert dict_keys to a list
country_keys = list(country.keys())

# Loop over list of keys; print associated values
for key in country_keys:
    print(country[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.

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

# Convert to a list
country_values = list(country.values()) # convert to a list

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

💡 You can also unpack a dictionary's values. Recall the Heritage site's `geo_coordinates` is a dictionary comprising the site's latitude and longitude. The values can be unpacked and assigned to variables.

```
# Unpacking
site_longitude, site_latitude = site['geo_coordinates'].values()

print(f"\n4.3.5 Site Geo coordinates = {site_longitude}, {site_latitude}")
```

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. Call `dict.items()` whenever you need to filter on specific keys in order to access a subset of the dictionary's values.

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

Challenge 01

Task: Convert the `country` dictionary's numeric values that are masquerading as strings to either an integer or a float.

1. In the `main` function block, loop over the `country` dictionary's items (keys and values) and utilize conditional logic to convert the following string values to either `int` or `float`:

Key	Convert value to
'population'	`int`
'surface_area_km2'	`int`
'urban_population_pct'	`float`

2. Uncomment the `print()` and `pp.pprint()` functions to check your output.

Challenge 02

Task: Call the two utility functions that utilize the `csv.DictReader` and `csv.DictWriter` objects to convert CSV rows to dictionaries and dictionaries to CSV rows.

💡 We will discuss these functions and the `csv.DictReader` and `csv.DictWriter` objects in more detail during the next lecture.

1. In the `main` function block, call the function `read_csv_to_dicts` and pass it the filepath `'./whc-sites-2022.csv'` as the lone argument. Assign the return value to a variable named `sites`.

   💡 The `sites` list contains 1155 dictionaries.

2. Create an empty accumulator list and assign it to a variable named `ukrainian_sites`. Loop over the `sites` list and in the loop block write a conditional statement that identifies Ukrainian World Heritage sites employing one of the following key-value pairs in your `if` statement:

Key	Value
'states_name_en'	'Ukraine'
'undp_code'	`ukr`

3. If the `if` statement evaluates to `True` append the dictionary to the `ukraine_sites` list.

4. After exiting the loop access the first dictionary in the `ukraine_sites` list and call its `.keys()` method. Assign the return value, a "view" object of the dictionary's keys, to a variable named `fieldnames`.

5. Call the function `write_dicts_to_csv` and pass it the following arguments in the specified order:

   1. `'whc-sites-ukraine-2022.csv'`
   2. `ukrainian_sites`
   3. `fieldnames`

   Confirm that the new CSV file was created in your lecture directory.