SI 506 Lecture 24

Topics

1. The Python module
    1. Importing modules
    2. Accessing module definitions and statements\
    3. Built-in `dir()` function
2. Sorting with the built-in function `sorted()` and `list.sort()` method (part II)
    1. Controlling the sort order with a `lambda` function
    2. Single condition sorting
    3. Multiple condition sorting
    4. Adding conditional logic to a `lambda` function
3. Challenges
    1. Challenge 01
    2. Challenge 02
    3. Challenge 03
    4. Challenge 04
    5. Challenge 05
    6. Challenge 06

Vocabulary

- **Cache**: Storage location that holds data in order in increase the speed by which previously requested data can be retrieved again if required. A cache is usually designed to hold data temporarily; cached data is allowed to "expire" after a given interval and is removed from storage. Cache expiration policies help to reduce the chance that data held in the cache no longer matches the origin data.
- **Higher order function**: a function that acts on or returns other functions. Built-in functions such as `map()`, `filter()`, and `sorted()` are consider higher-order functions as are the method `list.sort()` and the function `functools.reduce()`.
- **Key function**: A key function or collation function is a callable that returns a value used for sorting or ordering. Both `list.sort()` and the built-in function `sorted()` can be passed a key function such as a lambda as an optional argument in order to specify how a sequence is to be ordered.
- **Lambda**: An anonymous inline function consisting of a single expression which is evaluated when the function is called.
- **Module**: a Python file that contains definitions and statements that are intended to be *imported* into a Python script (a.k.a a program), an interactive console session, or another module.

Note on today's lecture code

Today's code features a basic caching implementation designed to reduce duplicate HTTP GET requests made to SWAPI. Resources retrieved from the web are stored locally in the cache. If a resource retrieved previously is again required it can be fetched from the local cache. The strategy optimizes for performance but care must be taken to ensure that cached representations of resources remain in sync with their remote origin counterparts otherwise the cache will grow stale. A cache that stores dynamic data (i.e., data that changes over time) must institute cache controls to ensure that locally stored resources are refreshed periodically. The caching code that you will encounter this week operates without an explicit cache expiration policy in order to keep the implementation simple.

Today's caching implementation involves the following objects:

Object	Module	Type	Description
`cache`	`lecture_24_utils.py`	`dict`	Serves as a temporary in-memory cache.
`create_cache_key`	`lecture_24_utils.py`	`function`	Formats a URL encoded string comprising the base URL, resource path, and querystring (if provided) that serves as a key to which the corresponding SWAPI resource is mapped.
`get_resource`	`lecture_24_utils.py`	`function`	Returns a dictionary representation of a JSON document retrieved from the web.
`get_swapi_resource`	`lecture_24.py`	`function`	Retrieves a dictionary representation of one or more SWAPI entities stored either remotely or locally in the `cache`. Resources retrieved via HTTP GET requests are stored in the local `cache`. The function delegates to `create_cache_key` the task of manufacturing keys to be used in the cache and to `get_resource` the task of retrieving remote resources.

💡 Caching will be discussed in more detail during the next lecture.

1.0 The Python module

Recall that Python features *two* file execution modes. Code in a file can be executed as a script from the command line or the code can be *imported* into another Python file in order to access its definitions and statements.

If a Python file is executed from the command line the Python interpreter will run the file under the special name of `__main__` rather than the program's actual file name (e.g., `lecture_XX.py`). We refer to such a file as a *script* or a program.

A *module* is a Python file that contains definitions and statements that are intended to be *imported* into a Python script (a.k.a a program), an interactive console session, or another module. If a Python file is imported as a *module* into another Python file it is known by its file name.

💡 Arguably, all Python files including scripts are modules since their definitions and statements can be imported into another Python file. But importing a Python script into another can result in unintended execution flow side effects so you need to think carefully about the purpose of each file you write as you modularize your code.

1.1 Importing modules

A module's definitions and statements can be *imported* into a Python program or script by referencing the module in an `import` statement.

💡 By convention `import` statements are located at the top of a Python file, although such placement is not required.

You've already learned that the [Python standard library](#) includes a number of "built-in" modules that must be explicitly imported in order to be used. Third-party libraries such as the `requests` package can also be installed and imported as modules.

```
import csv
import json
import requests
```

Local Python files can also be imported as modules. The filename less the `.py` extension constitutes the module name.

```
lecture_24.py           <-- intended as a script
lecture_24_utils.py     <-- intended as a module
```

💡 Review the module `lecture_24_utils.py`. Note that it imports other modules including `json` and `requests`.

```
import lecture_24_utils
```

Aliasing imported modules is accomplished using the reserved word (also called a *keyword*) `as` and specifying a short alias that is easy to comprehend:

```
import lecture_24_utils as utl
```

💡 Aliasing modules is recommended practice whenever working with a module name that is longer than 4-5 characters.

You can also import a module's definitions and statements directly using the `from` keyword:

```
from lecture_24_utils import SWAPI_ENDPOINT, get_swapi_resource,
read_json, write_json
```

```
from lecture_24_utils import (
    convert_data, convert_to_float, convert_to_int, get_swapi_resource,
    SWAPI_ENDPOINT, SWAPI_CATEGORES, SWAPI_FILMS, SWAPI_PEOPLE,
SWAPI_PLANETS,
    SWAPI_SPECIES, SWAPI_STARSHIPS, SWAPI_VEHICLES, read_json, write_json
    ) # (...) permits import statement to be expressed accross multiple
lines
```

You are even permitted to use a wildcard (`*`) in order to import all definitions and statements in a module (except those whose names begin with an underscore (`_`)):

```
from lecture_24_utils import *
```

❗ Although convenient, employing a wildcard to import a module's definitions and statements is *not recommended* because it introduces an unknown set of names that may overlap statements and defined earlier. It's opaqueness also undermines code readability.

💡 You can access a module name by using dot notation to reference the "dunder" `__name__` value.

```
import lecture_24_utils as utl

module_name = utl.__name__
```

1.2 Accessing module definitions and statements

Employ dot notation (`.`) to access a module's definitions and statements.

In the example below dot notation is used in the importing script or module to access both the function `get_swapi_resource` and the constant `SWAPI_PLANETS` (a URL) contained in the module `lecture_24_utils`.

```python
import lecture_24_utils

response =
lecture_24_utils.get_swapi_resource(lecture_24_utils.SWAPI_PLANETS,
{'search': 'hoth'})
```

Given the module name's length use of an alias is recommended instead:

```python
import lecture_24_utils as utl

response = utl.get_swapi_resource(utl.SWAPI_PLANETS, {'search': 'hoth'})
```

As noted above, you can also import definitions and statements directly employing the `from` keyword. Note that this approach does *not* introduce the module name from which the names are drawn to the importing script or module.

```python
from lecture_24_utils import get_swapi_resource, SWAPI_PLANETS

response = get_swapi_resource(SWAPI_PLANETS, {'search': 'hoth'})
```

You can also bind aliases to the imported names:

```python
from lecture_24_utils import get_swapi_resource as get, SWAPI_PLANETS as
url

response = get(url, {'search': 'hoth'})
```

1.3 Built-in `dir()` function

The built-in `dir()` function can be used to return a list containing a module's definitions and statement names.

```python
utl_names = dir(utl)

print(f"\nutl module's names = {utl_names}")
```

2.0 Sorting with the built-in function `sorted()` and `list.sort()` method (part II)

Recall that a Python function or method can accept another function as an argument. Such functions are considered "higher-order" functions. Several built-in functions accept functions as arguments. These include `filter()`, `map()`, and `sorted()`. The standard library's `functools` module also illustrates the pattern in practice (e.g., `functools.reduce()`) as does the `list.sort()` method.

Both the built-in function `sorted()` and the `list.sort()` method define a `key` parameter that permits a *key function* to be passed as an argument in order to specify a custom sort order.

In an earlier lecture I demonstrated how to apply a custom sort using a user-defined function as is illustrated below.

❗ Recall that `list.sort()` performs an *in-place* operation that mutates the current list while implicitly returning `None`. The built-in function `sorted()` exhibits different behavior. It returns a *new* list based on the list passed to it and sorted according to the key function (if any) defined for it.

```
def sort_by_population(entity):
    """Tries to return an < entity > dictionary's population value
converted to
    an integer.

    WARN: If the < entity > population value cannot be converted to an
integer the
    function returns zero (0) to the caller.

    Parameters:
        entity (dict): dictionary to parse

    Returns:
        int: returns an integer if the original value can be cast to a
string;
             otherwise, returns zero (0).
    """

    try:
        return int(entity['population'])
    except:
        return 0

# Sort by population size (ascending order, smallest to largest)
planets_sorted = sorted(planets, key=sort_by_population) # name of
function only

# Sort by population size (descending order, largest to smallest)
planets_sorted = sorted(planets, key=sort_by_population, reverse=True)

# Sort in-place: list.sort() method
planets.sort(key=sort_by_population, reverse=True)
```

2.1 Controlling the sort order with a `lambda` function

You can also pass a `lambda` function to a higher-order function as an argument. A Python `lambda` function is an *anonymous* function (i.e., a function defined without recourse to the `def` keyword) that specifies one or more parameters and a single expression that acts on the arguments provided it. The syntax to create a `lambda` function is

```
lambda < parameter or comma-separated set of parameters >: < expression referencing parameter(s) >
```

❗ Lambda syntax *does not* include a `return` statement. A Lambda function returns the expression defined for it *not* the anticipated value. A lambda can be passed to another expression and can be assigned to a variable.

You can experiment with lambdas by starting a terminal session and running the Python interactive console (a.k.a Python shell). macOS users start the Python shell by typing "python3" at the prompt and pressing the return key; Windows users can start the shell by typing either `python -i` or `winpty python` pressing the return key.

macOS

```
$ python3
```

Windows (Git Bash)

```
$ python -i
```

After activating the Python shell create and call the following lambda functions:

```
>>> (lambda x: x * 10)(5) # expression returned first followed by a value passed to it
50

>>> y = lambda x: x % 2 # expression assigned to the variable y

>>> y(30) # call y passing the value 30 to it
0

>>> y(17)
1

>>> addup = lambda x: sum(x) # sum takes an iterable

>>> addup([6, 50, 100, 150, 200])
506
```

```
>>> splitter = lambda x: x.split()

>>> words = splitter("These aren't the droids you are looking for.")

>>> words
['These', "aren't", 'the', 'droids', 'you', 'are', 'looking', 'for.']

>>> words_len = sorted(words, key=lambda x: len(x), reverse=True)

>>> words_len
['looking', "aren't", 'droids', 'These', 'for.', 'the', 'you', 'are']

>>> words_len = sorted(words, key=lambda x: -len(x))
>>> words_len
['looking', "aren't", 'droids', 'These', 'for.', 'the', 'you', 'are']
```

2.2 Single condition sorting

Applying a single condition to a `lamba` informed sort is a straightforward operation once you are comfortable with the syntax. Below are examples that illustrate how to sort a list of names employing the built-in function `sorted()` and the `list.sort()` method.

```
people = [
    'Obi-Wan Kenobi',
    'Luke Skywalker',
    'Chewbacca',
    'Leia Organa',
    'Han Solo',
    'Rey',
    'Lando Calrissian',
    'Poe Dameron',
    'Yoda'
]

# Alphanumeric sort
people_sorted = sorted(people, key=lambda x: x)

# Alphanumeric sort reversed
people_sorted = sorted(people, key=lambda x: x, reverse=True)

# Alphanumeric sort on last name
people_sorted = sorted(people, key=lambda x: x.split()[-1])

# Alphanumeric in-place sort reversed
people.sort(key=lambda x: x.split()[-1])
```

2.3 Multiple condition sorting

You can craft a lambda expression that applies multiple conditions to a sort. Below are examples that illustrate how to sort a list of dictionaries employing the built-in function `sorted()` and the `list.sort()` method.

❗ Multiple conditions *must* be expressed as an n-item tuple.

```
people = [
   {'name': 'Obi-Wan Kenobi', 'force_sensitive': True},
   {'name': 'Rey', 'force_sensitive': True},
   {'name': 'Luke Skywalker', 'force_sensitive': True},
   {'name': 'Leia Organa', 'force_sensitive': True},
   {'name': 'Han Solo', 'force_sensitive': False},
   {'name': 'Yoda', 'force_sensitive': True},
   {'name': 'Chewbacca', 'force_sensitive': False},
   {'name': 'Lando Calrissian', 'force_sensitive': False},
   {'name': 'Poe Dameron', 'force_sensitive': False},
   {'name': 'Finn', 'force_sensitive': False}
]

# Sort by name
people_sorted = sorted(people, key=lambda x: x['name'])

# Sort by force_sensitive (False = 0; True = 1), name
people_sorted = sorted(people, key=lambda x: (x['force_sensitive'],
x['name']))

# Sort by force_sensitive, name
people_sorted = sorted(people, key=lambda x: (x['force_sensitive'],
x['name']), reverse=True)

# Bidirectional sort by force_sensitive (True (1) then False (0)), name
people_sorted = sorted(people, key=lambda x: (-x['force_sensitive'],
x['name']))
```

2.4 Adding conditional logic to a `lambda` function

You can also embed conditional logic in a lambda function. For example, a `lambda` expression can employs a *ternary* operator to evaluate a sort value .

Ternary operator

```
[value when True] if [expression] else [value when False]
```

For example, given a list of SWAPI planet dictionaries, sorting on population would trigger a runtime exception since certain planets, such as the ice planet Hoth, are assigned `None` as the population value if the JSON value is `null`.

```json
{
  "url": "https://swapi.py4e.com/api/planets/4/",
  "name": "Hoth",
  "orbital_period_days": 549.0,
  "diameter_km": 7200,
  "gravity_std": 1.1,
  "climate": [
    "frozen"
    ],
  "terrain": [
    "tundra",
    "ice caves",
    "mountain ranges"
    ],
  "population": null
}
```

If you attempt to sort the planet dictionaries by their population value you will trigger a runtime exception:

```python
planets = sorted(planet_data, key=lambda x: x['population'])
```

```
...
TypeError: '<' not supported between instances of 'NoneType' and 'int'
```

This issue can be overcome by using the ternary operator in the lambda expression in which `None` is evaluated as zero (`0`).

```python
planets = sorted(planet_data, key=lambda x: x['population'] if
x['population'] else 0)
```

Sorting the planets by population in reverse order can be handled as follows:

```python
planets = sorted(planet_data, key=lambda x: x['population'] if
x['population'] else 0, reverse=True)
```

or

```python
planets = sorted(planet_data, key=lambda x: -x['population'] if
x['population'] else 0)
```

Bi-directional sorting involving population (descending order) and name (ascending order) is expressed as a two-item tuple in the `lambda` expression:

```
planets = sorted(planet_data, key=lambda x: (-x['population'] if
x['population'] else 0, x['name']))
```

3.0 Challenges

The following six challenges involve sorting a list of nine Star War films using a `lambda` function to control the sort order. Each JSON object that represents a Star Wars film is structured as follows:

```
{
    "url": "https://swapi.py4e.com/api/films/1/",
    "title": "A New Hope",
    "opening_crawl": "It is a period of civil war. . . .",
    "director": "George Lucas",
    "producers": [
      "Gary Kurtz"
    ],
    "release_date": "1977-05-25",
    "episode": 4
  }
```

The films are found in the file `films.json`.

3.1 Challenge 01

Task: Sort the films by episode number in descending order.

1. Import the module `lecture_24_utils.py` and assign it an alias named `utl`.

2. Read the file `films.json` by calling the utility module's function `read_json()`. Assign the return value to a variable named `film_data`.

3. Employ the build-in function `sorted()`, `film_data`, and a lambda expression to return a new list of the film dictionaries sorted by episode number in *descending* order. Assign the return value to variable named `films`.

4. Call the utility module's function `write_json()` and write the `films` list to a file named `stu_films-v1p0.json`.

5. Review the file. The list dictionaries must be ordered by episode number in descending order (i.e., 9, 8, 7, ..., 1).

3.2 Challenge 02

Task: Sort the films by release year in ascending order.

1. Employ the build-in function `sorted()`, `film_data`, and a lambda expression to return a new list of the film dictionaries sorted by the release **year** in *ascending* order. Assign the return value to variable named `films`.

   ❗ Extract the year value from each film's "release_date" key-value pair:

   ```
   {
    ...
      'release_date': '2019-12-20',
    ...
   }
   ```

   💡 There are at least two ways to solve this challenge. One approach is to embed a call to the `datetime.strptime()` method inside the lambda expression in order to access the each film's year value.

2. Call the utility module's function `write_json()` and write the `films` list to a file named `stu_films-v1p1.json`.

3. Review the file. The list of dictionaries *must* be ordered by year in ascending order (i.e., '1977-05-25', '1980-05-21', ..., '2019-12-20').

3.3 Challenge 03

Task: Sort the films by 1) director's last name (ascending order) and 2) the episode number (ascending order).

1. Employ the build-in function `sorted()`, `film_data`, and a lambda expression to return a new list of the film dictionaries sorted per the following conditions:

   1. director's **last name** in ascending order
   2. episode number in ascending order

   Assign the return value to variable named `films`.

   ❗ Extract the director's last name from each film's "director" key-value pair:

   ```
   {
    ...
      'director': 'J. J. Abrams',
    ...
   }
   ```

2. Call the utility module's function `write_json()` and write the `films` list to a file named `stu_films-v1p2.json`.

3. Review the file. The list of dictionaries *must* be ordered by director's last name, then by episode number in ascending order (i.e., 'J. J. Abrams' (episode 7), 'J. J. Abrams' (episode 9), ..., 'Richard Marquand' (episode 6)).

3.4 Challenge 04

Task: Sort the films by 1) director's last name (ascending order) and 2) the release date year (descending order).

1. Employ the build-in function `sorted()`, `film_data`, and a lambda expression to return a new list of the film dictionaries sorted per the following conditions:

    1. director's **last name** in ascending order
    2. release date year in **descending** order

    Assign the return value to variable named `films`.

💡 Now would be a good time to implement a bidirectional sort by applying the minus (`-`) sign to one of the conditions.

1. Call the utility module's function `write_json()` and write the `films` list to a file named `stu_films-v1p3.json`.

2. Review the file. The list of dictionaries *must* be ordered by director's last name, then by the release year in descending order (i.e., 'J. J. Abrams' (2019-12-20), 'J. J. Abrams' (2015-12-18), ..., 'Richard Marquand' (1983-05-25)).

3.5 Challenge 05

Task: Sort the films by 1) the number of producers (descending order) and 2) the episode number (descending order).

1. Employ the build-in function `sorted()`, `film_data`, and a lambda expression to return a new list of the film dictionaries sorted per the following conditions:

    1. number of producers in descending order
    2. episode number in **descending** order

    Assign the return value to variable named `films`.

2. Call the utility module's function `write_json()` and write the `films` list to a file named `stu_films-v1p4.json`.

3. Review the file. The list of dictionaries *must* be ordered by size of the producers list (descending order), then by the episode number (descending order) (i.e., ['Kathleen Kennedy', 'J. J. Abrams',

'Michelle Rejwan'] (episode 9), ['Kathleen Kennedy', 'J. J. Abrams', 'Bryan Burk'] (episode 7), ... ['Rick McCallum'] (episode 1)).

3.6 Challenge 06

Task: Perform an in-place sort of the `film_data` list, ordering the films by the release year in descending order.

1. Call the `film_data` list's `sort` method and perform an in-place sort, passing to the method as the key function a lambda expression that sorts the films by the release year. The film dictionaries *must* be sorted in **descending** order.

   💡 There are at least three ways to solve this challenge.

2. Call the utility module's function `write_json()` and write the `film_data` list to a file named `stu_films-v1p5.json`.

3. Review the file. The list of dictionaries *must* be ordered by size of the producers list (descending order), then by the episode number (descending order) (i.e., 'The Rise of Skywalker' (2019-12-20), 'The Last Jedi' (2017-12-15), ... 'A New Hope' (1977-05-25)).