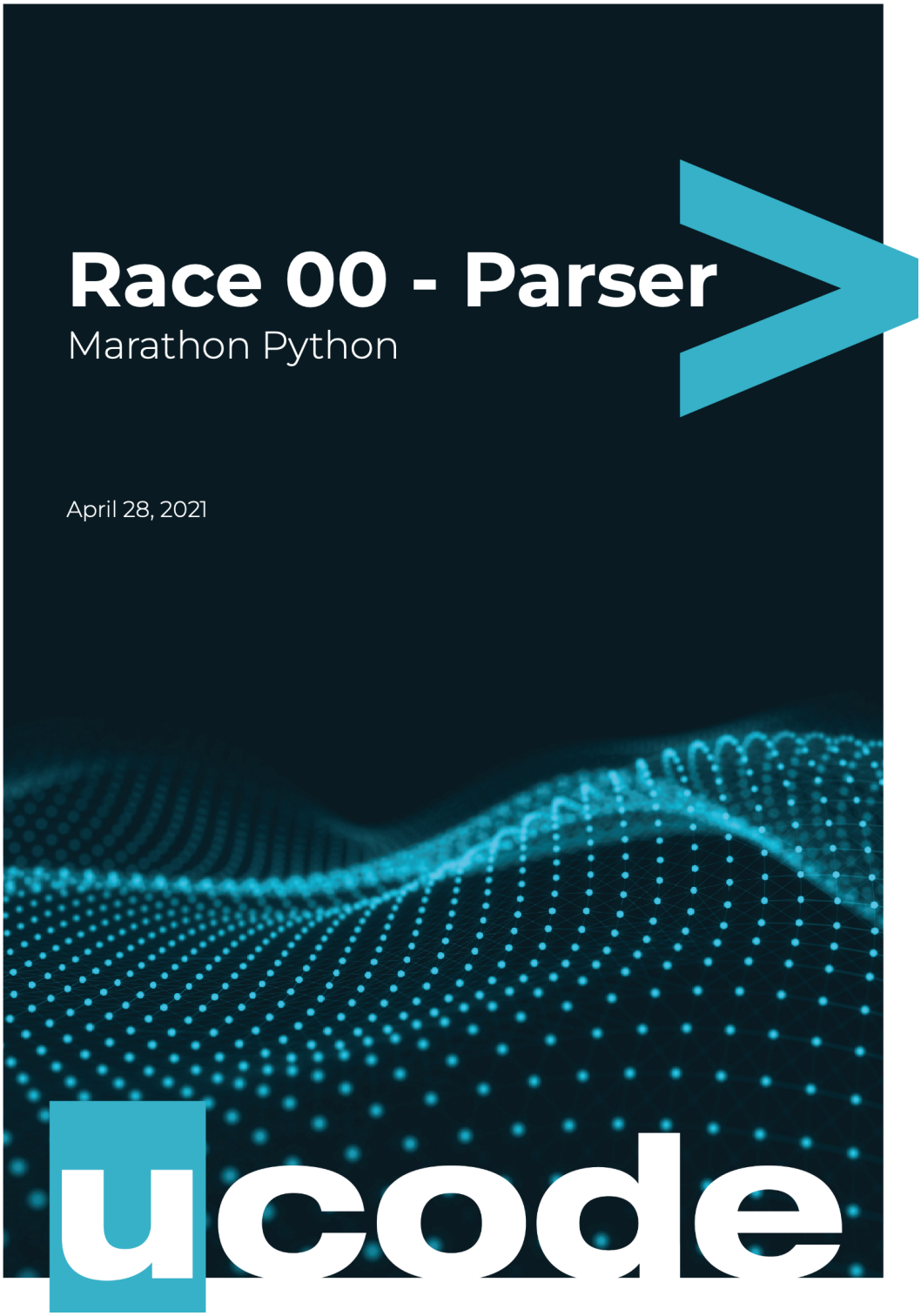


Contents

Engage

DESCRIPTION

Welcome to the Race - your first team challenge!!!

Teamwork skills are extremely important in life. Overcoming the challenges of working in a team, you will gain experience that will be useful to you in any field. Communication, conflict management, listening, reliability, respectfulness are all about teamwork.

Teamwork isn't as easy as it seems.
Be ready.
Keep calm and work as a team.

What will you face?
Programmers are often faced with tasks whose solution is not always obvious.
One of such tasks is parsing. In programming, very often you have to process huge amounts of information.
We are talking about such scales that sometimes manual selection is not just difficult - it is impossible: sometimes you need to collect thousands, or even millions of records.

In such cases, a special tool called a parser is usually used to collect and display information.
In this challenge, you will create a program that parses and visualizes serialized data.

Good luck, creativity and patience!

BIG IDEA

Create a data viewer program working as a team.

ESSENTIAL QUESTION

How to establish teamwork and correctly distribute tasks?

CHALLENGE

Manage your teamwork to succeed with the solution.

ucode

Race 00 - Parser | Marathon Python **> 2**

Investigate

GUIDING QUESTIONS

We invite you to find answers to the following questions. By researching and answering them, you will gain the knowledge necessary to complete the challenge. To find answers, ask the students around you and search the internet. We encourage you to ask as many questions as possible. Note down your findings and discuss them with your peers.

- Have you worked in a team before? What was your teamwork experience?
- What makes a successful team?
- What team-building activities do you know? Did you ever do any?
- What skills and strengths are required to become an effective team member?
- What tools can help you improve your teamwork skills?
- How to use `git` when you write code in collaboration? What gitflow do you use?
- What is a parser?
- How is a JSON file structured?
- What types of data can be passed to JSON?
- What is YAML and how is it different from JSON?
- How is a YAML file structured?
- How to load data from a YAML file?
- Why does the YAML loading method also work (most of the time) on JSON?
- What are the pros and cons of using the YAML loading method on JSON?
- What is recursion?
- In what situations are loops not enough, and recursion becomes the better solution?
- What are the different ways to visualize JSON or YAML data?
- What is the purpose of the `argparse` module?
- What is GUI?
- How to open a window using the module `Tkinter`?

GUIDING ACTIVITIES

Complete the following activities. Don't forget that you have a limited time to overcome the challenge. Use it wisely. Distribute tasks correctly.

- Meet with your team. Discuss how you will organize teamwork. Create a channel for team communication.
- Read about how to be an effective teammate.
- Research what skills you need for great teamwork.
- Read the story, taking everyone's ideas on board.
- Revise how to work with files in Python.
- Explore the features of working with JSON and YAML files.
- Look for inspiration by googling JSON viewers.

Race 00 - Parser | Marathon Python **> 3**

- Get to know Tkinter. Use the scripts given in the resources.
- Distribute tasks between all team members.
- Create a workplan for this challenge. Make sure the entire team understands what they need to do.
- Choose a gitflow for effective teamwork. We recommend you to use `git-flow-avh`.
- Clone your git repository that is issued on the challenge page in the LMS.
- Start to develop the solution. Offer improvements. Test your code.
- Communicate with students and share information.

ANALYSIS

Analyze your findings. What conclusions have you made after completing guiding questions and activities? In addition to your thoughts and conclusions, here are some more analysis results.

- The challenge has to be carried out by the entire team.
- You are working as a team. In programming, knowledge sharing is very important, so don't be afraid to ask your team members for review.
- Each team member must understand the challenge and realization, and must be able to reproduce it individually.
- It is your responsibility to assemble the whole team. Phone calls, SMS, messengers are good ways to stay in touch.
- You can proceed to `Act: Creative` only after you have completed all requirements in `Act: Basic`. But before you begin to complete the challenge, pay attention to the program's architecture. Take into account the fact that many features indicated in the `Act: Creative` require special architecture. And in order not to rewrite all the code somewhen, we recommend you initially determine what exactly you will do in the future.
- Be attentive to all statements of the story.
- Analyze all information you have collected during the preparation stages. Try to define the order of your actions.
- Perform only those tasks that are given in this document.
- Submit only those files that are described in the story. Only useful files allowed, garbage shall not pass!
- Run the scripts using `python3`.
- Make sure that you have `Python` with a `3.8` version, or higher.
- Use the standard library available after installing `Python`. You may use additional packages/libraries that were not previously installed only if they are specified in the challenge description.
- To figure out what went wrong in your code, use PEP 553 -- Built-in breakpoint().
- Complete the challenge according to the rules specified in the PEP8 conventions.
- The solution will be checked and graded by students like you. Peer-to-Peer learning.

Race 00 - Parser | Marathon Python **> 4**

- Also, the challenge will pass automatic evaluation which is called `Oracle`.
- If you have any questions or don't understand something, ask other students or just Google it.

ucode

Race 00 - Parser | Marathon Python **> 5**

Act: Basic

NAME

Parser

DIRECTORY

`./`

SUBMIT

`parser.py, get_data.py, *.py, README.md, testfiles/*`

DESCRIPTION

Create a data viewing GUI program that loads serialized data from a file, and visualizes the data in several different ways. The key idea is to be able to see the data in a more informative and human-readable way.

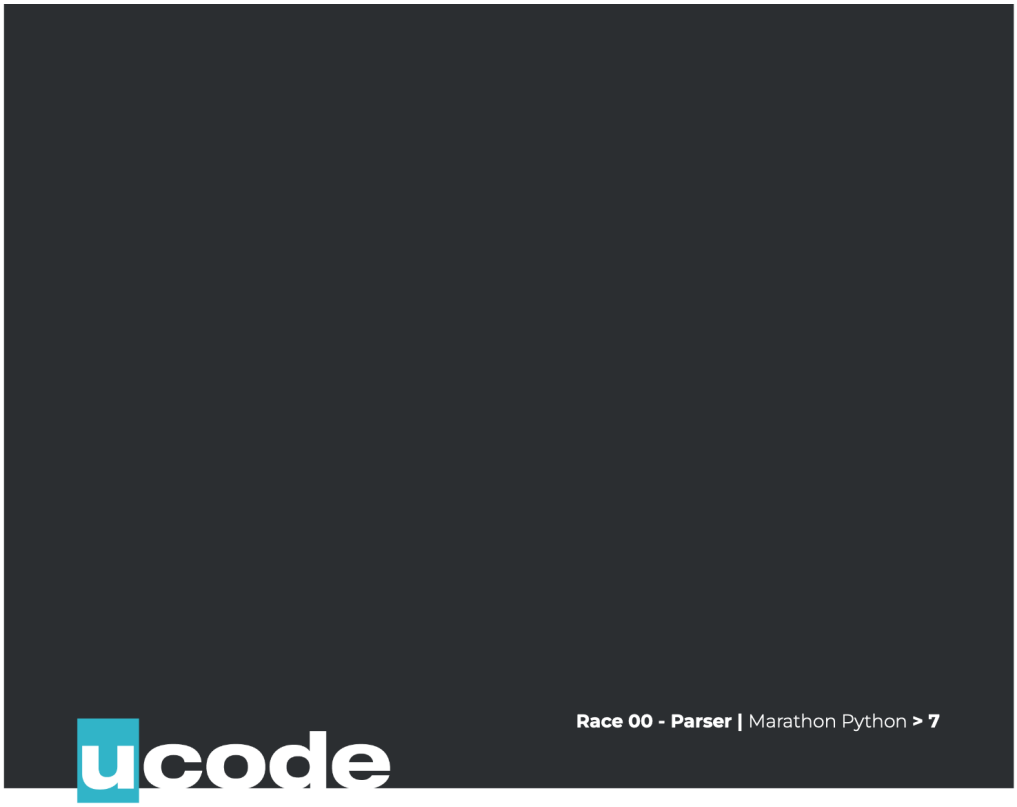
General requirements

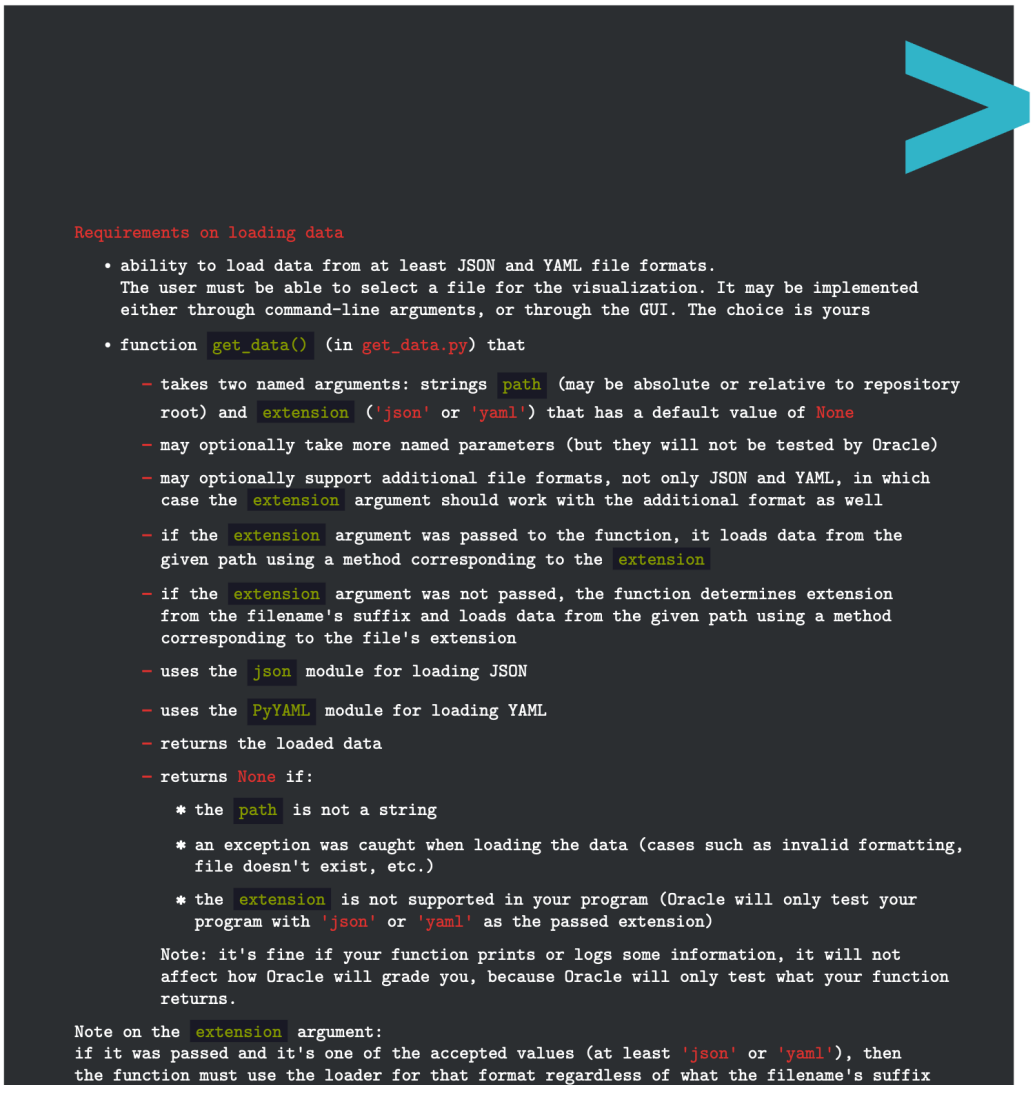
- submit:
  - `parser.py` as the main runnable script
  - `get_data.py` module that contains the `get_data()` function (described further in the story)
  - (optionally) more Python scripts that act as modules and are imported by `parser.py`
  - a detailed `README.md` that describes how to use your program, how to install necessary tools, etc. (more requirements further in the story)
  - (optionally) a directory `testfiles` with sample files (JSON, YAML, etc.) to test/demonstrate the work of your program (add your own data samples as well, not only the ones given in the `resources`)
- graphical user interface
- no crashes
- no uncaught exceptions
- no unexpected program exit
  The program must only exit in two scenarios:
  - couldn't launch because of invalid data given in the command-line arguments (if they are used in your program)
    (for example, your program expects a flag specifying what view to launch, and, if it wasn't provided, prints a message with the program's usage and exits)
  - the user chooses to exit (by closing the GUI window, selecting a corresponding menu option, etc.)

If at some point during the work of the program the user enters some invalid input, or attempts an action that isn't allowed in your program, this must not result in program exit.
Here's an example (just an example, not guidelines) of handling unexpected events without terminating the program:

Race 00 - Parser | Marathon Python **> 6**

Imagine your program allows the user to select the file to be viewed during the runtime of the program. The user selects a file that has invalid formatting. Instead of exiting, the program prints (or displays within the window) a message notifying that the file is not valid, and prompts the user to select another file.





Requirements on loading data

- ability to load data from at least JSON and YAML file formats.
  The user must be able to select a file for the visualization. It may be implemented either through command-line arguments, or through the GUI. The choice is yours
- function `get_data()` (in `get_data.py`) that
  - takes two named arguments: strings `path` (may be absolute or relative to repository root) and `extension` (`'json'` or `'yaml'`) that has a default value of `None`
  - may optionally take more named parameters (but they will not be tested by Oracle)
  - may optionally support additional file formats, not only JSON and YAML, in which case the `extension` argument should work with the additional format as well
  - if the `extension` argument was passed to the function, it loads data from the given path using a method corresponding to the `extension`
  - if the `extension` argument was not passed, the function determines extension from the filename's suffix and loads data from the given path using a method corresponding to the file's extension
  - uses the `json` module for loading JSON
  - uses the `PyYAML` module for loading YAML
  - returns the loaded data
  - returns `None` if:
    * the `path` is not a string
    * an exception was caught when loading the data (cases such as invalid formatting, file doesn't exist, etc.)
    * the `extension` is not supported in your program (Oracle will only test your program with `'json'` or `'yaml'` as the passed extension)

    Note: it's fine if your function prints or logs some information, it will not affect how Oracle will grade you, because Oracle will only test what your function returns.

Note on the `extension` argument:
if it was passed and it's one of the accepted values (at least `'json'` or `'yaml'`), then the function must use the loader for that format regardless of what the filename's suffix

is. For example, if 'yaml' was passed as the extension, use a yaml loading method, even if the filename is something like 'myfile.json' or 'myfile'. Also, if the extension argument was not passed, and the filename doesn't have a suffix that signals a certain supported file format (e.g., '.json' or '.yaml'), then just return None, don't try to guess the file format.

Examples of use of the function `get_data()`:
(the error messages that you can see in this is section are not obligatory, what matters is the returned data).

ucode

Race 00 - Parser | Marathon Python **> 8**

PYTHON INTERPRETER

```
>python3
>>> from get_data import get_data
>>> # file doesn't exist
>>> result = get_data(path='file_doesnt_exist', extension='json')
Error:  [Errno 2] No such file or directory: 'file_doesnt_exist'
>>> print(result)
None
>>> # error in loading (because trying to load yaml file with json loader)
>>> result = get_data(path='testfiles/sweets.yaml', extension='json')
Error:  Expecting value: line 1 column 1 (char 0)
>>> print(result)
None
>>> # using correct extension
>>> result = get_data(path='testfiles/sweets.yaml', extension='yaml')
>>> print(result)
[{'id': '0001', 'type': 'donut', 'name': 'Cake', 'ppu': 0.55, 'batters': {'batter':
↪  [{'id': '1001', 'type': 'Regular', 'something': True}, {'id': '1002', 'type':
↪  'Chocolate', 'something': None}, {'id': '1003', 'type': ['Blueberry', 'Chocolate']},
↪  {'id': '1004', 'type': "Devil's Food"}]}, 'topping': [{'id': '5001', 'type': 'None'},
↪  {'id': '5002', 'type': 'Glazed'}, {'id': '5005', 'type': 'Sugar'}, {'id': '5007',
↪  'type': 'Powdered Sugar'}, {'id': '5006', 'type': 'Chocolate with Sprinkles'}, {'id':
↪  '5003', 'type': 'Chocolate'}, {'id': '5004', 'type': 'Maple'}]}, {'id': '0001',
↪  'type': 'donut', 'name': 'Cake', 'ppu': 0.55, 'batters': {'batter': [{'id': '1001',
↪  'type': 'Regular'}, {'id': '1002', 'type': 'Chocolate'}, {'id': '1003', 'type':
↪  'Blueberry'}, {'id': '1004', 'type': "Devil's Food"}]}, 'topping': [{'id': '5001',
↪  'type': 'None'}, {'id': '5002', 'type': 'Glazed'}, {'id': '5005', 'type': 'Sugar'},
↪  {'id': '5007', 'type': 'Powdered Sugar'}, {'id': '5006', 'type': 'Chocolate with
↪  Sprinkles'}, {'id': '5003', 'type': 'Chocolate'}, {'id': '5004', 'type': 'Maple'}]}]
>>> # same case, but without specifying extension
>>> result = get_data(path='testfiles/sweets.yaml')
>>> print(result)
[{'id': '0001', 'type': 'donut', 'name': 'Cake', 'ppu': 0.55, 'batters': {'batter':
↪  [{'id': '1001', 'type': 'Regular', 'something': True}, {'id': '1002', 'type':
↪  'Chocolate', 'something': None}, {'id': '1003', 'type': ['Blueberry', 'Chocolate']},
↪  {'id': '1004', 'type': "Devil's Food"}]}, 'topping': [{'id': '5001', 'type': 'None'},
↪  {'id': '5002', 'type': 'Glazed'}, {'id': '5005', 'type': 'Sugar'}, {'id': '5007',
↪  'type': 'Powdered Sugar'}, {'id': '5006', 'type': 'Chocolate with Sprinkles'}, {'id':
↪  '5003', 'type': 'Chocolate'}, {'id': '5004', 'type': 'Maple'}]}, {'id': '0001',
↪  'type': 'donut', 'name': 'Cake', 'ppu': 0.55, 'batters': {'batter': [{'id': '1001',
↪  'type': 'Regular'}, {'id': '1002', 'type': 'Chocolate'}, {'id': '1003', 'type':
↪  'Blueberry'}, {'id': '1004', 'type': "Devil's Food"}]}, 'topping': [{'id': '5001',
↪  'type': 'None'}, {'id': '5002', 'type': 'Glazed'}, {'id': '5005', 'type': 'Sugar'},
↪  {'id': '5007', 'type': 'Powdered Sugar'}, {'id': '5006', 'type': 'Chocolate with
↪  Sprinkles'}, {'id': '5003', 'type': 'Chocolate'}, {'id': '5004', 'type': 'Maple'}]}]
>>> # trying to load file without specifying extension, and without suffix extension
>>> result = get_data(path='testfiles/sweets_without_ext')
Wrong file format.
>>> print(result)
None
>>> # same file, but extension specified
```

ucode

Race 00 - Parser | Marathon Python **> 9**

```
>>> result = get_data(path='testfiles/sweets_without_ext', extension='json')
>>> print(result)
[{'id': '0001', 'type': 'donut', 'name': 'Cake', 'ppu': 0.55, 'batters': {'batter':
↪  [{'id': '1001', 'type': 'Regular', 'something': True}, {'id': '1002', 'type':
↪  'Chocolate', 'something': None}, {'id': '1003', 'type': ['Blueberry', 'Chocolate']},
↪  {'id': '1004', 'type': "Devil's Food"}]}, 'topping': [{'id': '5001', 'type': 'None'},
↪  {'id': '5002', 'type': 'Glazed'}, {'id': '5005', 'type': 'Sugar'}, {'id': '5007',
↪  'type': 'Powdered Sugar'}, {'id': '5006', 'type': 'Chocolate with Sprinkles'}, {'id':
↪  '5003', 'type': 'Chocolate'}, {'id': '5004', 'type': 'Maple'}]}, {'id': '0001',
↪  'type': 'donut', 'name': 'Cake', 'ppu': 0.55, 'batters': {'batter': [{'id': '1001',
↪  'type': 'Regular'}, {'id': '1002', 'type': 'Chocolate'}, {'id': '1003', 'type':
↪  'Blueberry'}, {'id': '1004', 'type': "Devil's Food"}]}, 'topping': [{'id': '5001',
↪  'type': 'None'}, {'id': '5002', 'type': 'Glazed'}, {'id': '5005', 'type': 'Sugar'},
↪  {'id': '5007', 'type': 'Powdered Sugar'}, {'id': '5006', 'type': 'Chocolate with
↪  Sprinkles'}, {'id': '5003', 'type': 'Chocolate'}, {'id': '5004', 'type': 'Maple'}]}]
>>> # some more examples of loading results:
>>> get_data(path='testfiles/days.json')
['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']
>>> get_data(path='testfiles/colors.json')
{'red': {'hex': '#FF0000', 'rgb': 'rgb(255, 0, 0)'}, 'green': {'hex': '#008000', 'rgb':
↪  'rgb(0, 128, 0)'}, 'blue': {'hex': '#0000FF', 'rgb': 'rgb(0, 0, 255)'}, 'teal':
↪  {'hex': '#008080', 'rgb': 'rgb(0, 128, 128)'}, 'purple': {'hex': '#800080', 'rgb':
↪  'rgb(128, 0, 128)'}, 'yellow': {'hex': '#FFFF00', 'rgb': 'rgb(255, 255, 0)'},
↪  'black': {'hex': '#00000', 'rgb': 'rgb(0, 0, 0)'}}
>>> get_data(path='testfiles/movie.json')
{'title': 'Life of Brian', 'released': 1979, 'director': 'Terry Jones'}
>>>
```

The script get_data.py must not import any non-standard modules (apart from pyyaml).

Views

The main objective of your program is to visualize JSON data in a user-friendly way. You must implement at least two views, a tree view and a table view. There is more information on the views in the next two sections.

Tree view

Your program must be able to visualize data in a tree view. This type of view presents data in an hierarchical way. In the requirements below, we refer to items that (when loaded to Python) are lists or dictionaries as 'containers'.
Here are the requirements for this type of visualization:

- Each item of the data must be represented as a node.
- If an item is a container, it must be possible to expand/collapse its node to show/hide its sub-items.
- Container nodes must show whether they are a list or a dictionary. For example, you could add a prefix of either {} or [].
- Items of a list must show their index (starting from 0) within the list.

ucode

Race 00 - Parser | Marathon Python **> 10**

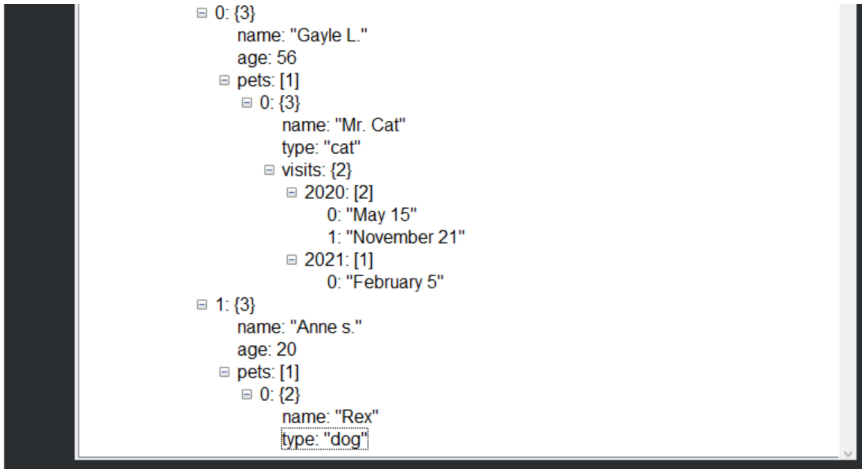
- The data type of the scalar values must be understandable from the visualization. The types of data to be distinguishable are strings, numerical (int or float), boolean, None. The most simple solution would be to use formatting: put quotation marks around the strings. But you're free to come up with your own system, as long as it's user-friendly.

- The nodes must be properly indented to represent the hierarchy.
- Your program must be able to visualize any level of nested data. Create a recursive function to achieve that. Make sure all members of your team understand how recursion works in general, and how it's implemented in your solution.

Below is an example of the tree view appearance with the file very_nested.json given in the resources. The widget used for this implementation is ttk.Treeview.

ucode

Race 00 - Parser | Marathon Python **> 11**



ucode

Race 00 - Parser | Marathon Python **> 12**

Table view

Your program must be able to visualize data in a table view. This type of visualization is best suited for data separated into multiple entries with repeating fields.

You are only required to handle cases with homogenous containers (all items similar). The cases you have to handle:

- list of scalar values (example with `days.json`)
  The first column must contain the item indexes within the list, and the second - the values. Keep in mind that even within the same list, the values may be of different scalar data types (string, integer, etc.). Whether the column with values is called 'value' or something else is up to you, as long as it's understandable.

```
[
  "Monday",
  "Tuesday",
  "Wednesday",
  "Thursday",
  "Friday",
  "Saturday",
  "Sunday"
]
```

	value
0	"Monday"
1	"Tuesday"
2	"Wednesday"
3	"Thursday"
4	"Friday"
5	"Saturday"
6	"Sunday"

- dictionary of scalar values (example with `movie.json`)
  The first column must contain the keys of the dict, and the second - the values. The values may be of different scalar data types.

```
{
  "title": "Life of Brian",
  "released": 1979,
  "director": "Terry Jones"
}
```

	value
title	"Life of Brian"
released	1979
director	"Terry Jones"

- list of dictionaries of scalar values (example with products.json)
  The first column must be item indexes in the list. Other columns must contain values for all the keys of the items-dictionaries. If some of the dicts don't have a value for a certain key, leave the cell blank, or use some placeholder string/symbol.
  The values within the same column may be of different scalar data types.

```
[
  {
    "id": "023",
    "title": "orange juice",
    "quantity": 54,
    "category": "beverages",
    "available": true
  },
  {
    "id": "407",
    "title": "strawberry yogurt",
    "quantity": 12,
    "category": "dairy",
    "available": true
  },
  {
    "id": "115",
    "title": "banana",
    "quantity": 0,
    "category": "fruits",
    "available": false
  },
  {
    "id": "437",
    "title": "milk",
    "category": "dairy",
    "available": true
  },
  {
    "id": "90",
    "title": "eggplant",
    "quantity": 10,
    "category": "vegetables",
    "available": true
  }
]
```

	id	title	quantity	category	available
0	"023"	"orange juice"	54	"beverages"	True
1	"407"	"strawberry yogurt"	12	"dairy"	True
2	"115"	"banana"	0	"fruits"	False
3	"437"	"milk"		"dairy"	True
4	"90"	"eggplant"	10	"vegetables"	True

- dictionary of dictionaries of scalar values (example with colors.json)
  The first column must contain the keys of the dictionary. Other columns must contain values for all the keys of the items-dictionaries (the entries of the outer dictionary).
  If some of the dicts don't have a value for a certain key, leave the cell blank, or use some placeholder string/symbol.
  The values within the same column may be of different scalar data types.

```
{
  "red": {
    "hex": "#FF0000",
    "rgb": "rgb(255, 0, 0)"
  },
  "green": {
    "hex": "#008000",
    "rgb": "rgb(0, 128, 0)"
  },
  "blue": {
    "hex": "#0000FF",
    "rgb": "rgb(0, 0, 255)"
  },
  "teal": {
    "hex": "#008080",
    "rgb": "rgb(0, 128, 128)"
  },
  "purple": {
    "hex": "#800080",
    "rgb": "rgb(128, 0, 128)"
  },
  "yellow": {
    "hex": "#FFFF00",
    "rgb": "rgb(255, 255, 0)"
  },
  "black": {
    "hex": "#00000",
    "rgb": "rgb(0, 0, 0)"
  }
}
```

ucode

Race 00 - Parser | Marathon Python **> 15**

	hex	rgb
red	"#FF0000"	"rgb(255, 0, 0)"
green	"#008000"	"rgb(0, 128, 0)"
blue	"#0000FF"	"rgb(0, 0, 255)"
teal	"#008080"	"rgb(0, 128, 128)"
purple	"#800080"	"rgb(128, 0, 128)"

yellow	"#FFFF00"	"rgb(255, 255, 0)"
black	"#00000"	"rgb(0, 0, 0)"

We encourage you to improve your program further and make it able to handle more complex cases, but it's not mandatory.

Even though you are not required to produce a visualization for other, more complex, cases, make sure your program does not crash if a user attempts to load such a file with the table view.

GUI

Your program must visualize data with a graphical user interface (GUI). On launch, a window must open. The visualization must be within that GUI window, not in the console. Use Tkinter as the GUI framework for your program. You're allowed to use Tkinter's additional modules.

There are several Python scripts available in the resources for the challenge. These scripts provide implementation examples and snippets of some basics of Tkinter. Whether to follow these examples or not is entirely up to you.

Even though this challenge requires GUI, don't get sidetracked by graphics and trying to make your program beautiful. Focus primarily on the logic of your solution, and try to complete all the minimum requirements before trying to add extra GUI features and styles.

Tools

You can use libraries and frameworks outside of the Standard Library. However, there's one limitation: all graphical rendering must be done through Tkinter and its modules.

If you will be using command-line arguments to specify options for your program (e.g., file path, extension, view option, etc.), we recommend that you use the `argparse` module for parsing the arguments.

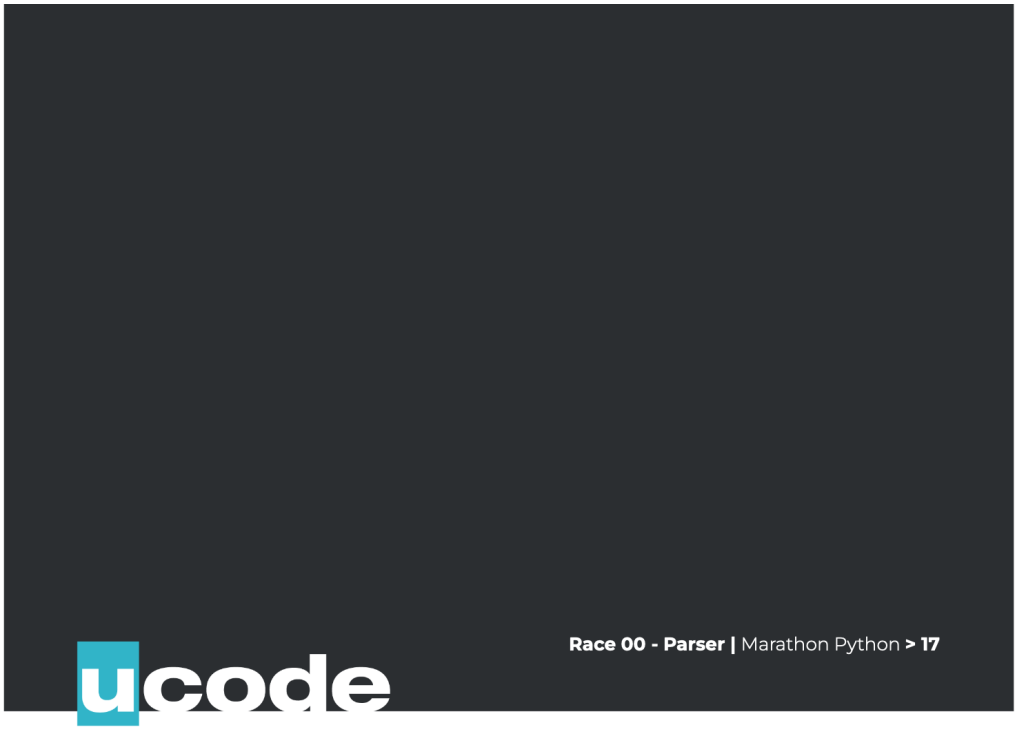
README

Your README.md file must contain the following sections:

- Name
  The name of your project
- Description
  Write a short paragraph describing your project.
- Prerequisites
  List of what needs to be installed/set up (and what version) for your program to run,

and how to do it.

- Usage
  Explain how to run your program, what arguments it may take. Show console examples of all the options.
  Include screenshots of your GUI in different scenarios.
- Authors
  List the people who have worked on the project and in what roles. Add links to their profiles on relevant sites.

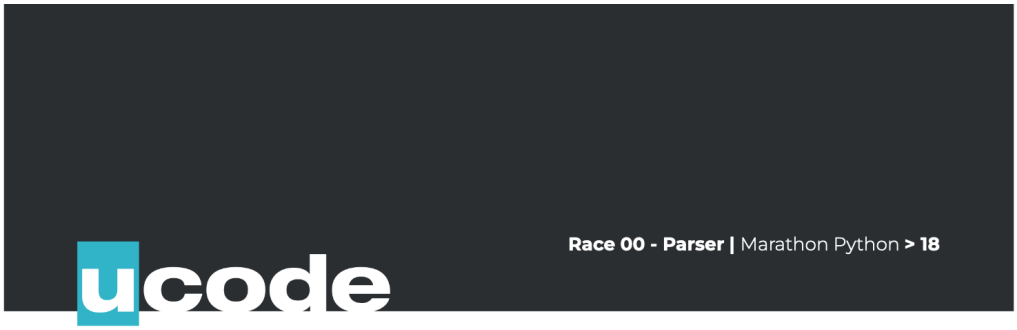


Act: Creative

DESCRIPTION

It is the place where your imagination and creativity plays a major role. Implement additional features to make the program better and more unique. Listed below are a few ideas you can add to your program. You can come up with everything you want to improve your program. Creative features:

- Support for additional data serialization file formats.
- Additional view of the data. Must be visually and conceptually different from tree and table. Code view/editor does not count.
- Nested table (tables within table cells to represent nested data). At least 5 levels.
- Editing within the view. Ability to edit data within the GUI tree or table view (e.g., add/delete/edit/move nodes). Must be possible to save the changes.
- Text editor mode for the data. For a particular file format or all of them. Must be possible to save the changes.
- Search within the data. A search mechanic that either highlights or displays a list of search results.
- Ability to filter data based on certain criteria.
- Sorting of table data by different columns.
- Multiple file support. Ability to visualize data from several files at the same time, e.g., in side-by-side frames. Does not count as an extra view.
- GUI file dialog to determine the file to be visualized.
- Ability to change the file during program runtime.
- Ability to update the contents of the selected file (to pull up changes made to the file after it was loaded). Selecting the file again doesn't count as this feature, there has to be a dedicated button/keyboard shortcut. Make sure to handle errors carefully.
- Other creative features.



Share

PUBLISHING

Last but not least, the final stage of your work is to publish it. This allows you to share your challenges, solutions, and reflections with local and global audiences. During this stage, you will discover ways of getting external evaluation and feedback on your work. As a result, you will get the most out of the challenge, and get a better understanding of both your achievements and missteps.

To share your work, you can create:

- a text post, as a summary of your reflection
- charts, infographics or other ways to visualize your information
- a video, either of your work, or a reflection video
- an audio podcast. Record a story about your experience
- a photo report with a small post

Helpful tools:

- Canva - a good way to visualize your data
- QuickTime - an easy way to capture your screen, record video or audio

Examples of ways to share your experience:

- Facebook - create and share a post that will inspire your friends
- YouTube - upload an exciting video
- GitHub - share and describe your solution
- Telegraph - create a post that you can easily share on Telegram
- Instagram - share photos and stories from ucode. Don't forget to tag us :)

Share what you've learned and accomplished with your local community and the world. Use #ucode and #CBLWorld on social media.