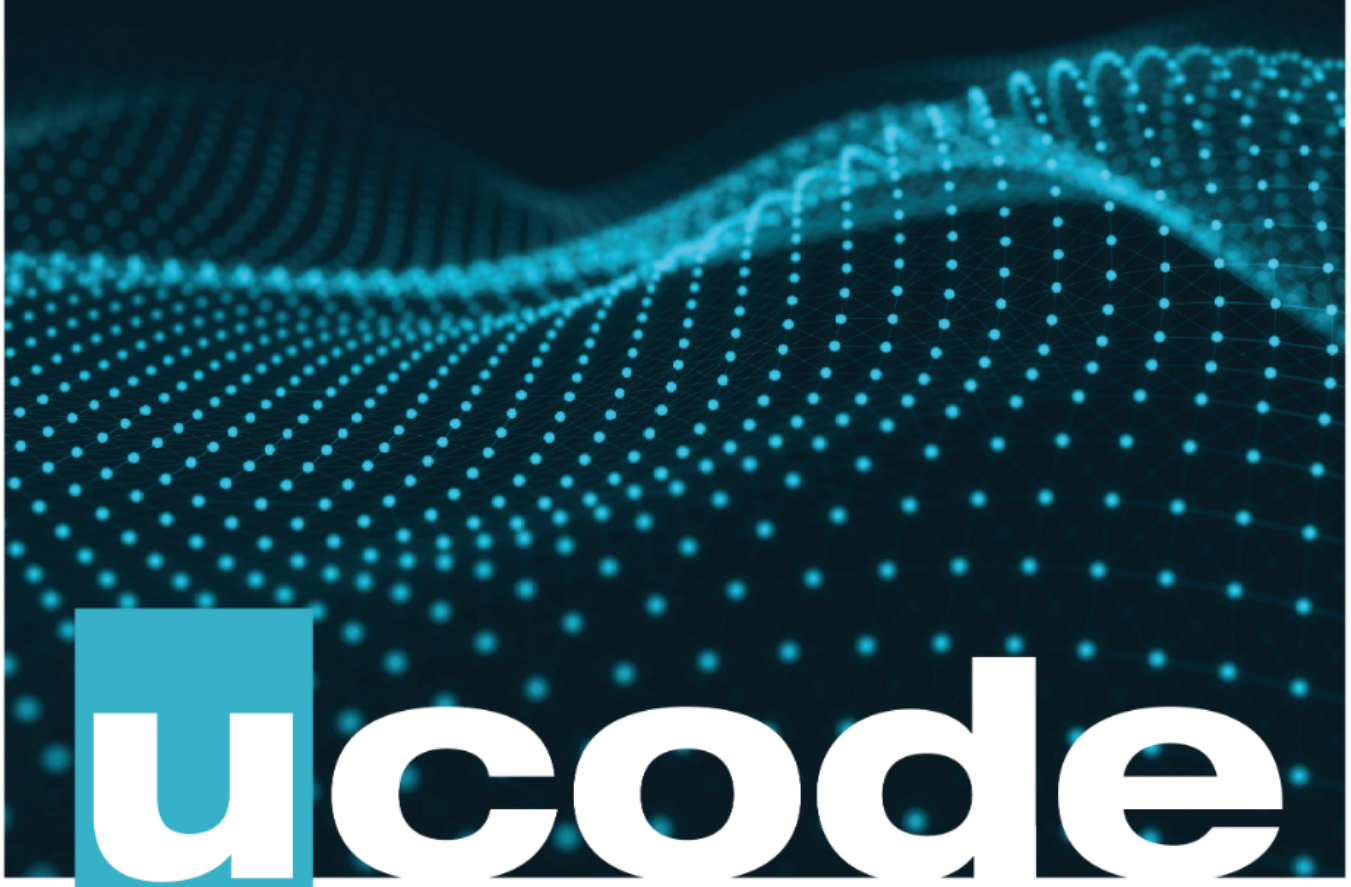
Sprint 03

Marathon Python

May 5, 2021



ucode

Contents

ucode

Engage

DESCRIPTION

Hello!

There are many different programming paradigms. For example, imperative, object-oriented, procedural, etc. One such paradigm is functional programming, inspired by Lambda Calculus.

Functional programming embraces mathematical-style computation using functions and avoids changing state and data mutability. Functional paradigm focuses on *what the code does*, compared to *how the code does it* (imperative programming).

Python is not a purely functional language, but it does contain certain features that allow programming in a functional style. One such feature that you will be learning in this challenge is lambda functions.

Also in this Sprint, you will learn how to work with files in Python. How to open, close, write information in them, and receive it.
You will also learn how to work with the JSON format.

So... Let's go?!

BIG IDEA

Functional programming and work with files.

ESSENTIAL QUESTION

How to write functional-style programs in Python?
How to interact with files?

CHALLENGE

Learn functional-style tools in Python.
Learn the basics of working with files in Python

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.

- What are the main concepts of functional programming?
- What are the benefits of functional programming?
- What are the features of functional programming in Python?
- What is a file in programming?
- What file formats are there?

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.

- Read about Python lambdas.
- Here you can find more info around lambdas in general.
- Learn how to read and write to files. For example, read this article.
- Familiarize yourself with JSON.
- Attentively watch and investigate learning videos available on the challenge page. Try to repeat all actions.
- Clone your git repository issued on the challenge page in the LMS.
- Proceed with tasks.

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.

- Be attentive to all statements of the story.
- All tasks are divided into Act Basic and Act Advanced. You need to complete all basic tasks to validate the Sprint. But to achieve maximum points, consider accomplishing advanced tasks also.
- 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 task.
- To figure out what went wrong in your code, use PEP 553 -- Built-in breakpoint().
- Complete tasks according to the rules specified in the PEP8 conventions.
- The solution will be checked and graded by students like you. Peer-to-Peer learning.
- 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.

Act Basic: Task 00

NAME

Lambda

DIRECTORY

`t00_lambda/`

SUBMIT

`expression.py`

BEFORE YOU BEGIN

In order to complete this task, you must be able to answer the following questions:

- What is a lambda function in Python?
- What is another name for lambda functions and why?
- What are the main features and limitations of lambda functions?
- What is the keyword used to create a lambda function?
- What does the following expression do `lambda a, b: a + b`? What will be the result if you pass in the values `(2, 2)`?

LEGEND

ENCYCLOPEDIA SALESMAN: Burglar! *[rings again]* Burglar!
[woman appears at other side of door]
WOMAN: Yes?
ENCYCLOPEDIA SALESMAN: Burglar, madam.
WOMAN: What do you want?
ENCYCLOPEDIA SALESMAN: I want to come in and steal a few things, madam.
WOMAN: Are you an encyclopaedia salesman?
ENCYCLOPEDIA SALESMAN: No madam, I'm a burglar, I burgle people.
WOMAN: I think you're an encyclopaedia salesman.
ENCYCLOPEDIA SALESMAN: Oh I'm not, open the door, let me in please.
WOMAN: If I let you in, you'll sell me encyclopedias.
ENCYCLOPEDIA SALESMAN: I won't, madam. I just want to come in and ransack the flat. Honestly.
WOMAN: Promise? No encyclopedias?
ENCYCLOPEDIA SALESMAN: None at all.
WOMAN: All right. *[she opens door]* You'd better come in then.
ENCYCLOPEDIA SALESMAN: Mind you, I don't know whether you've really considered the advantages of owning a really fine set of modern encyclopedias... You know, they can really do you wonders.
-- Monty Python's Flying Circus

DESCRIPTION

Create a script that contains a lambda function.
The lambda function must return `True` or `False` based on whether `n` is divisible by both `a`

and `b` with a remainder of 0.

Use the code snippet in the SYNOPSIS, but instead of ... put the needed code to get the correct results.
See examples of the script working in the CONSOLE VIEW.

SYNOPSIS

```
n = int(input('n: '))
a = int(input('a: '))
b = int(input('b: '))

# Only edit the following line
result = (lambda ...)(...)

print(result)
```

CONSOLE VIEW

```
>python3 expression.py
n: 15
a: 3
b: 5
True
>python3 expression.py
n: 64
a: 2
b: 14
False
>python3 expression.py
n: 10
a: 4
b: 5
False
>
```

SEE ALSO

Python lambda

Act Basic: Task 01

NAME

Clear words

DIRECTORY

`t01_clear_words/`

SUBMIT

`clear_words.py`

BEFORE YOU BEGIN

In order to complete this task, you must be able to answer the following questions:

- What does the Python `map()` function do?
- What parameters does the function `map()` take?
- How to use lambda expressions with `map()`?

DESCRIPTION

Create a function `clear_words()` that:

- takes some text as a string with punctuations marks. Punctuation marks must be `[?!.:;,-]` (use regex)
- splits the string by spaces
- uses the function `map()` with a lambda expression to remove all punctuation marks in the split words
- returns the result as a list

The script in the EXAMPLE tests your function. If everything is correct, it should generate output as seen in the CONSOLE VIEW. Also, you can see examples in the PYTHON INTERPRETER. Pay attention that you must only submit the file clear_words.py, not the test script.

EXAMPLE

```
from clear_words import clear_words


text_example_1 = 'WOMAN: Yes?, ENCYCLOPEDIA SALESMAN: Burglar, madam. WOMAN: '\
                 'Are you an encyclopaedia salesman?'

text_example_2 = 'ENCYCLOPEDIA SALESMAN: No madam, I\'m a burglar, '\
                 'I burgle people. WOMAN: I think you\'re an encyclopaedia '\
                 'salesman.'

text_example_3 = 'ENCYCLOPEDIA SALESMAN: Oh I\'m not, open the door, '\
                 'let me in please. WOMAN: If I let you in, you\'ll sell '\
                 'me encyclopedias.'
```

```
if __name__ == '__main__':
    result = clear_words(text_example_1)
    print(result)

    result = clear_words(text_example_2)
    print(result)

    result = clear_words(text_example_3)
    print(result)
```

CONSOLE VIEW

```
>python3 s03t01_clear_words_main.py
['WOMAN', 'Yes', 'ENCYCLOPEDIA', 'SALESMAN', 'Burglar', 'madam', 'WOMAN', 'Are', 'you',
↪  'an', 'encyclopaedia', 'salesman']
['ENCYCLOPEDIA', 'SALESMAN', 'No', 'madam', "I'm", 'a', 'burglar', 'I', 'burgle',
↪  'people', 'WOMAN', 'I', 'think', "you're", 'an', 'encyclopaedia', 'salesman']
['ENCYCLOPEDIA', 'SALESMAN', 'Oh', "I'm", 'not', 'open', 'the', 'door', 'let', 'me',
↪  'in', 'please', 'WOMAN', 'If', 'I', 'let', 'you', 'in', "you'll", 'sell', 'me',
↪  'encyclopedias']
>
```

PYTHON INTERPRETER

```
>python3
>>> from clear_words import clear_words
>>> text_example_1 = 'Now, you listen here: \'e\'s not the Messiah, \'e\'s a very naughty
↪  boy! Now piss off!'
>>> result = clear_words(text_example_1)
>>> result
['Now', 'you', 'listen', 'here', "'e's", 'not', 'the', 'Messiah', "'e's", 'a', 'very',
↪  'naughty', 'boy', 'Now', 'piss', 'off']
>>> text_example_2 = 'There\'s a l|ot .of. punc::tuat!on h-e-r-e! And ,,,,, not @ll of
↪  ___ it#is corr>ct.'
>>> result = clear_words(text_example_2)
>>> result
["There's", 'a', 'l|ot', 'of', 'punctuaton', 'here', 'And', '', 'not', '@ll', 'of',
↪  '___', 'it#is', 'corr>ct']
>>> text_example_3 = ''
>>> result = clear_words(text_example_3)
>>> result
[]
>>> text_example_4 = '         '
>>> result = clear_words(text_example_4)
```

```
>>> result
[]
>>> text_example_5 = ',,,,,'
>>> result = clear_words(text_example_5)
>>> result
['']
>>>
```

SEE ALSO

Python - map() Function
Python Lambda Functions with Examples

Act Basic: Task 02

NAME

The extractor

DIRECTORY

`t02_the_extractor/`

SUBMIT

`extractor.py`

BEFORE YOU BEGIN

In order to complete this task, you must be able to answer the following questions:

- How does the method `filter()` work?
- What are the requirements for the function that is passed to `filter()`?
- What happens if you pass `None` instead of a function to `filter()`?
- Does `filter()` modify an iterable in-place, or return the result?
- How to check if a value is of a certain data type?
- Why does this expression `isinstance(False, int)` return `True`?

LEGEND

Now, you listen here: 'e's not the Messiah, 'e's a very naughty boy! Now piss off!
-- *Monty Python's Life of Brian*

DESCRIPTION

Create a script with a function `extractor()` that filters a dictionary according to a certain data type of the values. It takes two arguments: a dictionary `extractable` and a data type `value_type` (an instance of class `type`, e.g., `int`, `bool`, etc). `value_type` has the default value of `str`.

The function uses `filter()` to create a new dictionary that only contains the items from `extractable` that have a value of type `value_type`.

The script in the `EXAMPLE` tests your function. Use this as an example, add your own values, and also test your function as shown in the `PYTHON INTERPRETER`. If everything is correct, it should generate output as shown below. Pay attention that you must only submit the file `extractor.py`, not the test script, and Oracle will check your function with random values.

EXAMPLE

```
from extractor import extractor


if __name__ == '__main__':
    brian = {
        'name': 'Brian',
        'email': 'very_naughty_boy@gmail.com',
        'age': 33,
        'star_sign': 'Capricorn',
        'legs': 2,
        'arms': 2.5
    }
    print(f'***\nbrian, str: {extractor(brian, str)}')
    print(f'***\nbrian, int: {extractor(brian, int)}')
    print(f'***\nbrian, list: {extractor(brian, list)}')

    monty_python = {
        'title': 'Monty Python',
        'media': ['film', 'tv', 'radio', 'audio', 'literature'],
        'country of origin': 'United Kingdom',
        'first aired': 1969,
        'genres': ['satire', 'surreal', 'black comedy'],
        'key members': {'Graham Chapman',
                        'John Cleese',
                        'Terry Gilliam',
                        'Eric Idle',
                        'Terry Jones',
                        'Michael Palin'},
        'official website': 'montypython.com',
        'nb of films': 5,
        'nb of tv series': 3,
        'average sketch length': 30.5,
        'songs': {
            'The Meaning Of Life':
                ['Accountancy Shanty', 'Galaxy Song'],
            'Monty Python And The Holy Grail':
                ['The Tale Of Sir Robin', 'Camelot Song'],
            'Life Of Brian':
                ['Brian Song', 'Always Look On The Bright Side Of Life'],
            'Monty Python's Flying Circus':
                ['The Liberty Bell', 'Lumberjack Song', 'Spam Song']},
        'name has meaning': False,

    }
    print(f'***\nmonty_python, list:\n{extractor(monty_python, list)}')
    print(f'***\nmonty_python, float:\n{extractor(monty_python, float)}')
    print(f'***\nmonty_python, dict:\n{extractor(monty_python, dict)}')
    print(f'***\nmonty_python, no data type given:\n{extractor(monty_python)}')
```

```
    print(f'***\nempty, int:\n{extractor({}, dict)}')
```

CONSOLE VIEW

```
>python3 s03t02_the_extractor_main.py
***
brian, str: {'name': 'Brian', 'email': 'very_naughty_boy@gmail.com', 'star_sign':
↪  'Capricorn'}
***
brian, int: {'age': 33, 'legs': 2}
***
brian, list: {}
***
monty_python, list:
{'media': ['film', 'tv', 'radio', 'audio', 'literature'], 'genres': ['satire',
↪  'surreal', 'black comedy']}
***
monty_python, float:
{'average sketch length': 30.5}
***
monty_python, dict:
{'songs': {'The Meaning Of Life': ['Accountancy Shanty', 'Galaxy Song'], 'Monty Python
↪  And The Holy Grail': ['The Tale Of Sir Robin', 'Camelot Song'], 'Life Of Brian':
↪  ['Brian Song', 'Always Look On The Bright Side Of Life'], 'Monty Python's Flying
↪  Circus': ['The Liberty Bell', 'Lumberjack Song', 'Spam Song']}}
***
monty_python, no data type given:
{'title': 'Monty Python', 'country of origin': 'United Kingdom', 'official website':
↪  'montypython.com'}
***
empty, int:
{}
>
```

PYTHON INTERPRETER

```
>python3
>>> from extractor import extractor
>>> today = {'weekday': 'Friday', 'day': 13, 'month': 'July', 'year': 2099, 'weather':
↪  ['cloudy', 'radioactive rain'], 'full moon': False}
>>> extractor(today, int)
{'day': 13, 'year': 2099, 'full moon': False}
>>> extractor(today, str)
{'weekday': 'Friday', 'month': 'July'}
>>>
```

SEE ALSO

Python filter() function
type() vs. isinstance()

Act Basic: Task 03

NAME

Read file

DIRECTORY

`t03_read_file/`

SUBMIT

`read_file.py`

BEFORE YOU BEGIN

In order to complete this task, you must be able to answer the following questions:

- How to work with files in Python?
- What modes can you choose when opening a file with `open()`?
- Why is it important to close an opened file?
- What methods does Python have for reading files?
- What error will Python throw if you try to access a file that does not exist? What about a file that has no read permissions?

DESCRIPTION

Create a script that contains a function `read_file()` that:

- takes one argument: filename as a string
  (You are not required to handle cases when the argument is not of type `str`.)
- if the file is present in the current directory (and the user has read permissions for it):
  - opens the file
  - reads the file and saves its contents in a variable
  - prints the file contents in the following format:
    'File "[filename]" has the following message:
    [file contents]'
  - closes the file
- if there is no such file in the current directory (or the user has no read permissions for the requested file):
  - prints the message: 'Failed to read file "[filename]".'

The script in the EXAMPLE tests your function (using the files sample.txt, sample1.txt, and no_permission.txt which are available in the resources). If everything is correct, it should generate output as seen in the CONSOLE VIEW. Also, you can see examples in the PYTHON INTERPRETER.
Pay attention that you must only submit the file read_file.py, not the test script.

EXAMPLE

```
import os

from read_file import read_file

if __name__ == '__main__':
    strings = ['sample.txt', 'sample1.txt', 'no_permission.txt']
    # remove read permissions from one file to check error handling
    os.chmod('no_permission.txt', 000)
    for string in strings:
        read_file(string)
        print('***')
```

CONSOLE VIEW

```
>python3 s03t03_read_file_main.py
File "sample.txt" has the following message:
"You only killed the bride's father, you know."
"I didn't mean to."
"Didn't mean to? You put your sword right through his head."
"Oh dear... is he all right?"
***
File "sample1.txt" has the following message:
Bedevere: "What makes you think she is a witch?"
Peasant: "She turned me into a newt."
Bedevere: "A newt?"
Peasant: "Well I got better."
***
Failed to read file "no_permission.txt".
***
>
```

PYTHON INTERPRETER

```
>python3
>>> from read_file import read_file
>>> read_file('hello.txt')
File "hello.txt" has the following message:
Hello there, ucoder!
Let's open and read some files!
>>> read_file('nonexistent.txt')
Failed to read file "nonexistent.txt".
>>> read_file('goodbye.txt')
Goodbye!
You have learned something new today!
>>> import os
```

```
>>> os.access('without_permissions.txt', os.R_OK) # check if file has read permissions
True
>>> os.chmod('without_permissions.txt', 000) # remove read permissions
>>> os.access('without_permissions.txt', os.R_OK) # check read permissions again
False
>>> read_file('without_permissions.txt') # now try to read the file
Failed to read file "without_permissions.txt".
>>> read_file('empty.txt')
File "empty.txt" has the following message:

>>>
```

SEE ALSO

Reading and Writing Files
Python File Open
UNIX File Permission/Access Modes

Act Basic: Task 04

NAME

Write

DIRECTORY

`t04_write/`

SUBMIT

`write_file.py`

BEFORE YOU BEGIN

In order to complete this task, you must be able to answer the following questions:

- How to write in a file in Python?
- What happens if you use the `write()` method with `'w'` mode to write to a file that is not empty?
- What is a context manager?
- How does the `with` statement work?
- Why is the `with` statement particularly useful for working with files?
- How to check if an open file has been closed?

DESCRIPTION

Create a script that contains a function `write_file()`. This function:

- takes two arguments:
  - filename as a string
    If the given filename does not end with `.txt`, prints two consecutive messages:
    `'Please enter the filename in the format "filename.txt".'`
    `'Failed to write to file "[filename]".'`
  - message to be added
    (Assume that the function will receive only strings for both arguments. There is no need to handle other cases.)
    The default value of the message argument must be set to `'None'`.
- creates a file with the given name (if it doesn't exist yet) and opens it for writing
- writes the given message to the file and closes the file
- opens the file again to check if it is not empty
  - if the file is empty or its contents are not equal to the provided message, prints the message: `'Something went wrong...'`
  - otherwise, prints two consecutive messages:
    * `'Your message has been written to file "[filename]".'`
    * `'File "[filename]" now contains the following text:
      [message]'`

For this task, use the `with` statement in your solution.

The script in the EXAMPLE tests your function. If everything is correct, it should generate output as seen in the CONSOLE VIEW. Also, you can see examples in the PYTHON INTERPRETER. Pay attention that you must only submit the file write_file.py, not the test script.

EXAMPLE

```
from write_file import write_file

if __name__ == '__main__':

    write_file('file')
    print('***')
    write_file('file.txt')
    print('***')
    write_file('file', 'Hello there')
    print('***')
    write_file('new_file.txt', 'Hello there')
    print('***')
    # check with a non-empty file
    write_file('example.txt', 'A new message')
    print('***')
```

CONSOLE VIEW

```
>python3 s03t04_write_main.py
Please enter the filename in the format "filename.txt".
Failed to write to file "file".
***
Your message has been written to file "file.txt".
File "file.txt" now contains the following text:
None
***
Please enter the filename in the format "filename.txt".
Failed to write to file "file".
***
Your message has been written to file "new_file.txt".
File "new_file.txt" now contains the following text:
Hello there
***
Your message has been written to file "example.txt".
File "example.txt" now contains the following text:
A new message
***
>
```

PYTHON INTERPRETER

```
>python3
>>> from write_file import write_file
>>> write_file('new.txt')
Your message has been written to file "new.txt".
File "new.txt" now contains the following text:
None
>>> write_file('new')
Please enter the filename in the format "filename.txt".
Failed to write to file "new".
>>> write_file('file.txt', 'Some text')
Your message has been written to file "file.txt".
File "file.txt" now contains the following text:
Some text
>>> with open('example.txt', 'r') as f:
...     print(f'File contents before writing:\n{f.read()}')
...
File contents before writing:
This file is not empty!
>>> write_file('example.txt', 'A new message')  # write to a non-empty file
Your message has been written to file "example.txt".
File "example.txt" now contains the following text:
A new message!
>>>
```

SEE ALSO

Python with Statement

Act Basic: Task 05

NAME

Load JSON

DIRECTORY

`t05_load_json/`

SUBMIT

`summary.py`

BEFORE YOU BEGIN

In order to complete this task, you must be able to answer the following questions:

- What is JSON, and why is it useful?
- What module needs to be imported to work with JSON in Python?
- What are serialization and deserialization?
- How to load JSON data from a file? How is it different from loading from a string?
- What data type in JSON does a dict correspond to?
- What data types cannot be keys in a dict?
- How to sort a dict by values in reverse order?

DESCRIPTION

Create a script with a function `summary()` that returns a summary of value frequencies for a certain field in JSON data. The function takes a string `filename` and a value `summarize_by`, and returns a summary dict.

Firstly, load the JSON data stored in the file with the `filename`. In case the JSON is invalid, catch the exception and return a string
`'Error in decoding JSON.'`.

In this task, the JSON data in `filename` will always be an array of objects (when loaded to Python, becomes a list of dicts). You don't have to handle cases when it isn't. Also, you will be tested only with existing files, so handling cases when the file doesn't exist is not mandatory.

The main goal of the function is to answer the question:
given a list of data entries (the given JSON array of objects), what different values does this particular key (`summarize_by`) have and how common are they?
For example, if you have an online platform and want to find out what countries your clients are from, you could call:
`summary('users.json', 'country')`
and get a resulting dict looking something like this:
`'France': 1549, 'India': 870, 'Australia': 47`.
Side note: such a function is useful only when we expect the values to be recurring, like categories of something (e.g., movie genres, popcorn flavors), and would not be very

informative for values that are unique to an entry (e.g., email address), or have too much deviation (e.g., age, name).

Detailed requirements on creating the summary:

- The returned summary is a dict with values of `summarize_by` as keys, and their counts as values.
- The function goes through all dicts in the loaded JSON data list, and for every dict it performs the following logic:
  - look for an item with key `summarize_by`
  - if found: increase the counter of that value by one
  - if not found: increase the counter of None by one
  - if found, but the value is None, increase the counter of None by one
  - if found, but the value is unhashable, increase the counter of 'unhashable' by one
- The summary has to be sorted in descending order by values (from values that appear the most to the least common).

The script in the EXAMPLE tests your function. Use this as an example and add your own values. If everything is correct, it should generate output as shown below. Pay attention that you must only submit the file summary.py, not the test script, and Oracle will check your function with random values.

Note about counting value occurences:
If you want, you can try using a `Counter` dict subclass from the `collections` module. It's a powerful module from The Python Standard Library that is worth trying out. Also, the method `setdefault()` may be useful in your solution.

EXAMPLE

```
from summary import summary

if __name__ == '__main__':
    print(summary('s03t05_load_json_test.json', 'visibility'))
    print(summary('s03t05_load_json_test.json', 'purpose'))
    print(summary('s03t05_load_json_test.json', 'no items with this key'))
    print(summary('s03t05_load_json_invalid.json', 'purpose'))
```

CONSOLE VIEW

```
>python3 s03t05_load_json_main.py
{'public': 263, 'private': 237}
{'Promotion': 102, 'Entertainment': 82, None: 75, 'Connection': 62, 'Inspiration': 59,
↪  'Conversation': 56, 'Education': 56, '': 5, 'unhashable': 2, False: 1}
{None: 500}
Error in decoding JSON.
>
```

SEE ALSO

Python JSON Parsing using json.load() and loads()
JSON Data Types

Act Basic: Task 06

NAME

Dump JSON

DIRECTORY

`t06_dump_json/`

SUBMIT

`quad.py`

BEFORE YOU BEGIN

In order to complete this task, you must be able to answer the following questions:

- How to create a JSON string from a Python dict?
- How to make a JSON string indented?
- What's the difference between `json.dump()` and `json.dumps()`?
- What is a quadratic equation?
- Why can't `a` be zero in a quadratic equation?
- How to tell how many real solutions a quadratic equation will have from the value of the discriminant?
- How to check if a value is one of several data types with a single call to `isinstance()`?
- How to round a number to a certain number of decimal places?

DESCRIPTION

Create a script with a function `quad()` that takes three numbers (`a`, `b`, and `c`), and returns a JSON string with information about a quadratic equation with these values. The result must be indented with three levels.

If one of the arguments is not an integer or a float, or `a` is equal to `0`, then the function must return `'Cannot generate a quadratic equation.'`

The resulting object must have two fields: `"equation"` and `"solution"`. The value of `"equation"` must be a string representing the equation (see format in the CONSOLE VIEW). Pay close attention to the output format, especially to cases where the arguments are `1`, `-1`, or `0`.

The value of the `"solution"` must be an object with the fields: `"discriminant"`, with the value of the discriminant of the equation, and `"x"`, with the value of x for the equation. Both the discriminant and value(s) of x must be outputted as floats, rounded to three decimal places (rounding only affects the JSON output, not the calculations). Also, make sure you don't have any negative zeros in your output.

Your function must only calculate real solutions. In case the solutions are complex (hint: when the discriminant is negative), the value of `"x"` must be `null`. If there are two real solutions for x, the value of the field `"x"` must be a sorted array of the two solutions. Otherwise (one real solution), just a number with the value of x.

The script in the EXAMPLE tests your function. If everything is correct, it should generate output as shown below. Pay attention that you must only submit the file quad.py, not the test script, and Oracle will check your function with random values.

EXAMPLE

```
from quad import quad
from random import seed, randint


def run_test(a, b, c):
    print(f'---\n{a}, {b}, {c}')
    print(quad(a, b, c))


if __name__ == '__main__':
    run_test(1, 0, 0)
    run_test(-7.5, 0, 0)
    run_test(120, -1, 0)
    run_test(15, 1.8, 1)
    run_test(1, 4, 4)
    run_test(1, 5, 6)
    run_test('a', 5, 6)
    run_test(0, 1, 2)
    seed(5)
    for _ in range(3):
        run_test(randint(-20, 20), randint(-20, 20), randint(-20, 20))
```

CONSOLE VIEW

```
>python3 s03t06_dump_json_main.py
---
1, 0, 0
{
   "equation": "x^2=0",
   "solution": {
      "discriminant": 0.0,
      "x": 0.0
   }
}
---
-7.5, 0, 0
{
   "equation": "-7.5x^2=0",
   "solution": {
      "discriminant": 0.0,
      "x": 0.0
   }
```

```
}
---
120, -1, 0
{
   "equation": "120x^2-x=0",
   "solution": {
      "discriminant": 1.0,
      "x": [
         0.0,
         0.008
      ]
   }
}
---
15, 1.8, 1
{
   "equation": "15x^2+1.8x+1=0",
   "solution": {
      "discriminant": -56.76,
      "x": null
   }
}
---
1, 4, 4
{
   "equation": "x^2+4x+4=0",
   "solution": {
      "discriminant": 0.0,
      "x": -2.0
   }
}
---
1, 5, 6
{
   "equation": "x^2+5x+6=0",
   "solution": {
      "discriminant": 1.0,
      "x": [
         -3.0,
         -2.0
      ]
   }
}
---
a, 5, 6
Cannot generate a quadratic equation.
---
0, 1, 2
Cannot generate a quadratic equation.
---
```

```
19, -4, 2
{
   "equation": "19x^2-4x+2=0",
   "solution": {
      "discriminant": -136.0,
      "x": null
   }
}
---
13, -19, 9
{
   "equation": "13x^2-19x+9=0",
   "solution": {
      "discriminant": -107.0,
      "x": null
   }
}
---
-5, -17, -10
{
   "equation": "-5x^2-17x-10=0",
   "solution": {
      "discriminant": 89.0,
      "x": [
         -2.643,
         -0.757
      ]
   }
}
>
```

SEE ALSO

Python JSON dump() and dumps() for JSON Encoding
Quadratic Equations

Act Advanced: Task 07

NAME

Calculambdor

DIRECTORY

`t07_calculambdor/`

SUBMIT

`calculator.py`

DESCRIPTION

Create a script that contains a function `calculator()` and a global variable `operations`. The function takes three arguments: a string with a name of a mathematical operation and two numbers (`int` or `float`). The `calculator()` function returns the result of the given operation on the two numbers.
Available operations and their names:

- `+` - `add`
- `-` - `sub`
- `*` - `mul`
- `/` - `div`
- `**` - `pow`

`operations` must be a dictionary of lambda functions, where the operation names listed above are the keys, and the lambdas are the values. If the passed values are invalid, raise a `ValueError` with a relevant message:
'Invalid numbers. Second and third arguments must be numerical.' or
'Invalid operation. Available operations: add, sub, mul, div, pow.'.

Take a look at the PYTHON INTERPRETER section.

PYTHON INTERPRETER

```
>python3
>>> from calculator import calculator, operations
>>> calculator('add', 2, '2')
Traceback (most recent call last):
  File "<stdin>", line N, in <module>
  File "your_path", line N, in calculator
    raise ValueError('Invalid numbers. Second and third arguments must be numerical.')
ValueError: Invalid numbers. Second and third arguments must be numerical.
>>> calculator('add', 2, 2)
4
>>> calculator('addition', 2, 2)
Traceback (most recent call last):
  File "<stdin>", line N, in <module>
  File "your_path", line N, in calculator
```

```
    raise ValueError('Invalid operation. Available operations: add, sub, mul, div, pow.')
ValueError: Invalid operation. Available operations: add, sub, mul, div, pow.
>>> calculator('sub', 1000, 999)
1
>>> calculator('mul', 9, 2.4)
21.599999999999998
>>> calculator('div', 81, 9)
9.0
>>> calculator('pow', 2, 9)
512
>>> # testing the `operations` dict
>>> operations.keys()
dict_keys(['add', 'sub', 'mul', 'div', 'pow'])
>>> operations['add'].__name__
'<lambda>'
>>> callable(operations['div'])
True
>>> operations['mul'](6, -0.5)
-3.0
>>>
```

SEE ALSO

Raising Exceptions

Act Advanced: Task 08

NAME

Multiplier

DIRECTORY

`t08_multiplier/`

SUBMIT

`multiplier.py`

BEFORE YOU BEGIN

In order to complete this task, you must be able to answer the following questions:

- What is the role of the `all()` function?
- How does this function work for lists?
- What does the `functools` module do?
- What's the principle of the `reduce()` function?

DESCRIPTION

Create a script that contains a function called `multiplier()`.
It takes a list of numbers: `int` and/or `float`, and returns the result of their multiplication.

If the argument is not of type `list` or its elements are not all numerical (use the `all()` function), raise a `ValueError` with a message `'The given data is invalid.'`.

Use a lambda function in combination with the `reduce()` function in your implementation.
Import the appropriate module.

EXAMPLE

```
from multiplier import multiplier


if __name__ == '__main__':
    test_1 = [2.72, 43, 4]
    test_2 = [22, 3.04, 26]
    test_3 = [23, 5, 3.36]

    print(multiplier(test_1))
    print('***')

    print(multiplier(test_2))
    print('***')

    print(multiplier(test_3))
    print('***')
```

CONSOLE VIEW

```
>python3 s03t08_multiplier_main.py
467.84000000000003
***
1738.8799999999999
***
386.4
***
>
```

PYTHON INTERPRETER

```
>python3
>>> from multiplier import multiplier
>>> test_1 = [3, 8, 2.04]
>>> multiplier(test_1)
48.96
>>> test_2 = [2, 5, 7, -3]
>>> multiplier(test_2)
-210
>>> test_3 = ['string', 3, 5.1]
>>> multiplier(test_3)
Traceback (most recent call last):
  File "<stdin>", line N, in <module>
  File "your_path", line N, in multiplier
    raise ValueError('The given data is invalid.')
ValueError: The given data is invalid.
>>>
```

SEE ALSO

Python all()
functools
reduce() in Python

Act Advanced: Task 09

NAME

Raise

DIRECTORY

`t09_raise/`

SUBMIT

`raise_error.py`

BEFORE YOU BEGIN

In order to complete this task, you must be able to answer the following questions:

- How to raise an error?
- What kind of situations can lead to `EOFError`?
- What exception does it make sense to raise if someone passes a number to a function that expects a string: `ValueError` or `TypeError`? What about when a passed string doesn't meet some particular requirements (e.g. not empty, is capitalized, etc.)?
- Why is the message of a `KeyError` formatted differently from others?

DESCRIPTION

Create a script that contains a function called `raise_error()`. The function takes two strings: a key and a message, and raises an error corresponding to the given key with the message. If the key is not one of the following:

- `'value'`
- `'key'`
- `'index'`
- `'memory'`
- `'name'`
- `'eof'`

raise a `ValueError` with a message `'No error with such key.'`.

The script in the `EXAMPLE` tests your function. Use this as an example and add your own values. If everything is correct, it should generate output as shown in the `CONSOLE VIEW`. Pay attention that you must only submit the file `raise_error.py`, not the test script, and Oracle will check your function with random values.

EXAMPLE

```python
from raise_error import raise_error


if __name__ == '__main__':
    errors = ['value', 'key', 'index', 'memory', 'name', 'eof', 'wrong', '']
    for error in errors:
        message = f'This is a `{error}` error.'
        try:
            raise_error(error, message)
        except Exception as e:
            print(f'call with "{error}", "{message}":'
                  f'\n\tRaised error: {type(e)}'
                  f'\n\tMessage: {str(e)}\n')
```

CONSOLE VIEW

```
>python3 s03t09_raise_main.py
call with "value", "This is a `value` error.":
        Raised error: <class 'ValueError'>
        Message: This is a `value` error.

call with "key", "This is a `key` error.":
        Raised error: <class 'KeyError'>
        Message: 'This is a `key` error.'

call with "index", "This is a `index` error.":
        Raised error: <class 'IndexError'>
        Message: This is a `index` error.

call with "memory", "This is a `memory` error.":
        Raised error: <class 'MemoryError'>
        Message: This is a `memory` error.

call with "name", "This is a `name` error.":
        Raised error: <class 'NameError'>
        Message: This is a `name` error.

call with "eof", "This is a `eof` error.":
        Raised error: <class 'EOFError'>
        Message: This is a `eof` error.

call with "wrong", "This is a `wrong` error.":
        Raised error: <class 'ValueError'>
        Message: No error with such key.

call with "", "This is a `` error.":
        Raised error: <class 'ValueError'>
```

```
        Message: No error with such key.

>
```

SEE ALSO

Python - Error Types

Act Advanced: Task 10

NAME

Group

DIRECTORY

`t10_group/`

SUBMIT

`group.py`

LEGEND

SUPERMAN ONE: Oh look... is it a stockbroker?
SUPERMAN TWO: Is it a quantity surveyor?
SUPERMAN THREE: Is it a church warden?
ALL SUPERMEN: NO! It's Bicycle Repair Man!
-- Monty Python's Flying Circus

BEFORE YOU BEGIN

Research the itertools module in Python. This is a very powerful module that provides tools for more effective and complex iteration. In this task, you will need specifically the `groupby()` method. However, explore the module more and experiment with other methods as well, such as `count()`, `cycle()`, `product()`, etc.

DESCRIPTION

Create a script that contains a function called `group()`. This function takes two arguments: a dict `to_group` and a list `keys` of fields to group by. It returns a new dictionary that is grouped by the fields provided in the list of fields (in the order of the list).

See examples in the CONSOLE VIEW. The output was achieved by running the s03t10_group_main.py script in the EXAMPLE. Pay attention that you must only submit the file group.py, not the test script, and Oracle will check your function with random values.

EXAMPLE

```
import json
from group import group

if __name__ == '__main__':
    test_case_data = [
        {
            'name': 'Alex',
            'gender': 'male',
            'species': 'human',
            'job': 'bicycle repair man',
        },
        {
            'name': 'Monika',
            'gender': 'female',
            'species': 'human',
            'job': 'stockbroker'
        },
        {
            'name': 'Bob',
            'gender': 'male',
            'species': 'human',
            'job': 'quantity surveyor'
        },
        {
            'name': 'Veronika',
            'gender': 'female',
            'species': 'human',
            'job': 'church warden'
        },
        {
            'name': 'George',
            'gender': 'male',
            'species': 'human',
            'job': 'bicycle repair man'
        },
    ]
    test_case_data_group_fields_1 = ['gender']
    test_case_data_group_fields_2 = ['species', 'gender', 'job']

    res_1 = group(test_case_data, test_case_data_group_fields_1)
    res_2 = group(test_case_data, test_case_data_group_fields_2)

    print(json.dumps(res_1, indent=2))
    print()
    print(json.dumps(res_2, indent=2))
```

CONSOLE VIEW

```
>python3 s03t10_group_main.py
{
  "female": [
    {
      "name": "Monika",
      "species": "human",
      "job": "stockbroker"
    },
    {
      "name": "Veronika",
      "species": "human",
      "job": "church warden"
    }
  ],
  "male": [
    {
      "name": "Alex",
      "species": "human",
      "job": "bicycle repair man"
    },
    {
      "name": "Bob",
      "species": "human",
      "job": "quantity surveyor"
    },
    {
      "name": "George",
      "species": "human",
      "job": "bicycle repair man"
    }
  ]
}

{
  "human": {
    "female": {
      "church warden": [
        {
          "name": "Veronika"
        }
      ],
      "stockbroker": [
        {
          "name": "Monika"
        }
      ]
    },
    "male": {
      "bicycle repair man": [
```

```
        {
          "name": "Alex"
        },
        {
          "name": "George"
        }
      ],
      "quantity surveyor": [
        {
          "name": "Bob"
        }
      ]
    }
  }
}
>
```

SEE ALSO

groupby

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.