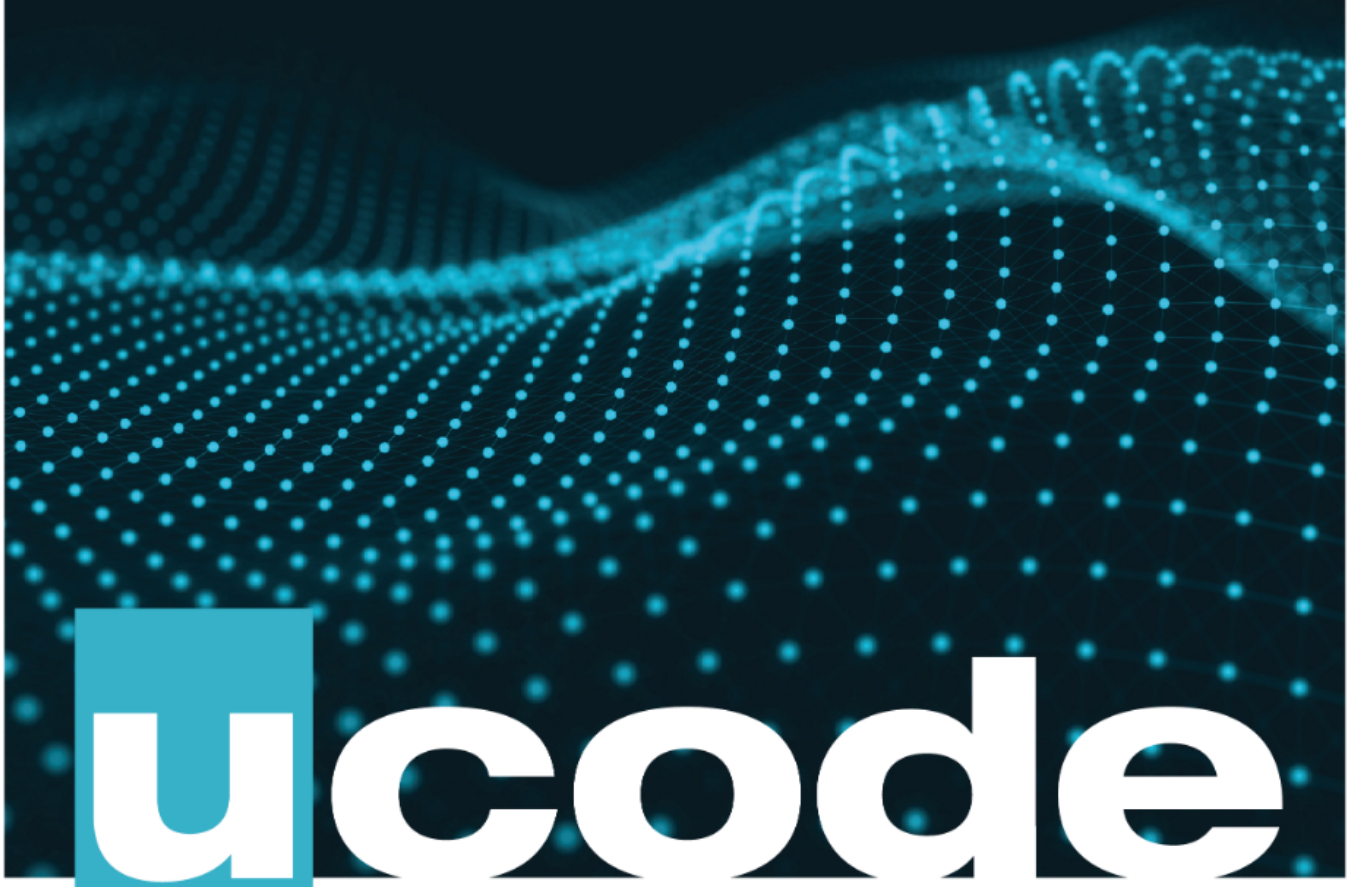
Sprint 02

Marathon Python

June 14, 2021



ucode

Contents

ucode

Engage

DESCRIPTION

Let's get started with Python data structures.

Data structures are a fundamental construct in programming. Each data structure provides a particular way of organizing data so it can be accessed efficiently, depending on use case.

For example, phone books make a decent real-world analog for dictionary objects. They allow you to quickly retrieve the information (phone number) associated with a given key (a person's name). Instead of having to read a phone book front to back to find someone's number, you can jump more or less directly to a name and look up the associated information.

Data structures provide us with a specific way of storing and organizing data, such that it can be easily accessed and worked with efficiently. There are quite a few data structures available. The Python built-in data structures are: lists, tuples, dictionaries, sets.

BIG IDEA

Data structures.

ESSENTIAL QUESTION

What are the ways to store data in Python?

CHALLENGE

Explore data structures and their appropriate uses.

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 is a data structure?
- What data structures exist in Python?
- What are the most popular sorting algorithms?
- What is a regular expression in Python?

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 built-in datatypes.
- Find information about sorting and sorting algorithms.
- Learn about regular expressions and how to use them in Python.
- In this online regex visualizer, build a regex that recognizes all words that start with an uppercase letter.
- 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

Tuple

DIRECTORY

`t00_tuple/`

SUBMIT

`song.py`

BEFORE YOU BEGIN

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

- What is a `tuple` in Python?
- How is a tuple different from a list?
- What is the advantage of using a tuple over a list?
- How to create a tuple with just one element?
- How to add tuples?
- How to multiply tuples?

DESCRIPTION

Create a script with a function `song()` that generates a song from given verses and chorus. It takes two tuple arguments: `verses` and `chorus`, and returns a tuple of strings. The `verses` is a tuple of tuples of strings, and the `chorus` is a tuple of strings. Each string is meant to be a line of a song.

The formula for a song that your function must use is: add a chorus after every verse, and add one more chorus in the end of the song. So, for example, if the given arguments are:

- `verses`: `(('a', 'b'), ('c', 'd'))`
- `chorus`: `('x', 'y')`

Then the function must return `('a', 'b', 'x', 'y', 'c', 'd', 'x', 'y', 'x', 'y')`. You can see this example in the PYTHON INTERPRETER.

The script in the EXAMPLE tests your function. If everything is correct, it should generate output as seen in the CONSOLE VIEW. Pay attention that you must only submit the file `song.py`, not the test script.

You don't have to manage test cases where the incoming arguments are not tuples.

EXAMPLE

```
from song import song


simple_verses = (('verse1 line1', 'verse1 line2'),
                 ('verse2 line1', 'verse2 line2', 'verse2 line3'),
                 ('verse3 line1',))
simple_chorus = ('** CHORUS line1', '** CHORUS line2')
love_verses = (("There's nothing you can do that can't be done",
                "Nothing you can sing that can't be sung",
                'Nothing you can say but you can learn how to play the game',
                "It's easy\n"),
               ("Nothing you can know that isn't known",
                "Nothing you can see that isn't shown",
                "Nowhere you can be that isn't where you're meant to be",
                "It's easy\n"))
love_chorus = ('All you need is love', 'All you need is love',
               'All you need is love, love', 'Love is all you need\n')


def print_test_case(verses, chorus, test_description):
    print('\n' + f'test - {test_description}'.center(40, '_'))
    print('[start]', *song(verses, chorus), '[end]', sep='\n')


if __name__ == '__main__':
    print_test_case(simple_verses, simple_chorus, 'simple')
    print_test_case(love_verses, love_chorus, 'normal song')
    print_test_case((tuple(), ('verse2 line1', 'verse2 line2')),
                    simple_chorus, 'one verse = empty tuple')
    print_test_case(tuple(), simple_chorus, 'verses = empty tuple')
    print_test_case(simple_verses, tuple(), 'chorus = empty tuple')
    print_test_case(tuple(), tuple(), 'both arguments empty')
```

CONSOLE VIEW

```
>python3 s02t00_tuple_main.py

_____________test - simple_______________
[start]
verse1 line1
verse1 line2
** CHORUS line1
** CHORUS line2
verse2 line1
verse2 line2
verse2 line3
** CHORUS line1
** CHORUS line2
verse3 line1
** CHORUS line1
** CHORUS line2
** CHORUS line1
** CHORUS line2
[end]

____________test - normal song____________
[start]
There's nothing you can do that can't be done
Nothing you can sing that can't be sung
Nothing you can say but you can learn how to play the game
It's easy

All you need is love
All you need is love
All you need is love, love
Love is all you need

Nothing you can know that isn't known
Nothing you can see that isn't shown
Nowhere you can be that isn't where you're meant to be
It's easy

All you need is love
All you need is love
All you need is love, love
Love is all you need

All you need is love
All you need is love
All you need is love, love
Love is all you need

[end]
```

```
_____test - one verse = empty tuple_____
[start]
** CHORUS line1
** CHORUS line2
verse2 line1
verse2 line2
** CHORUS line1
** CHORUS line2
** CHORUS line1
** CHORUS line2
[end]

______test - verses = empty tuple_______
[start]
** CHORUS line1
** CHORUS line2
[end]

______test - chorus = empty tuple_______
[start]
verse1 line1
verse1 line2
verse2 line1
verse2 line2
verse2 line3
verse3 line1
[end]

______test - both arguments empty_______
[start]
[end]
>
```

PYTHON INTERPRETER

```
>python3
>>> from song import song
>>> verses = (('a', 'b'), ('c', 'd'))
>>> chorus = ('x', 'y')
>>> song(verses, chorus)
('a', 'b', 'x', 'y', 'c', 'd', 'x', 'y', 'x', 'y')
>>>
```

SEE ALSO

Understanding Tuples in Python 3

Act Basic: Task 01

NAME

Dict

DIRECTORY

`t01_dict/`

SUBMIT

`cookbook.py`

BEFORE YOU BEGIN

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

- What is a `dict` in Python?
- What error would be raised if you tried to access a missing key in this way: `my_dict['no key like this']`?
- What is the advantage of using the method `get()`?
- What data types can be keys in a `dict`?

DESCRIPTION

Create a script with a function `search_cookbook()`. It takes three arguments: `cookbook`, `recipe`, and `section`. The `cookbook` is a dict of dicts. The function looks for an item in the `cookbook` dict with `recipe` as the key. If found, it then looks for an item in the `recipe` dict with `section` as the key. On success, the function returns the found item.

If there is no `recipe` in the `cookbook`, the function returns the string:
`'There is no "[recipe]" recipe in the cookbook.'`
If there is no item with `section` key in the `recipe`, the function returns the string:
`'There is no section "[section]" in the "[recipe]" recipe.'`

The function must use the method `get()`.

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

Note that you don't have to handle the scenario where the values inside the `cookbook` are not dicts. Assume that `cookbook` values can only be of type `dict`.

EXAMPLE

```
from cookbook import search_cookbook


def test_cookbook(target_cookbook, target_recipe, target_section):
    print(f'Searching for section "{target_section}" '
          f'in recipe "{target_recipe}":')
    print(search_cookbook(target_cookbook, target_recipe, target_section))
    print('---')


if __name__ == '__main__':
    cookbook = {
        'Salmon salad': {
            'ingredients': ['salmon', 'tomatoes', 'spinach'],
            'prep time': 15,
            'directions': 'Cut tomatoes. Mix all ingredients together.'
        },
        'Fried eggs': {
            'ingredients': ['eggs', 'salt', 'black pepper'],
            'prep time': 5
        },
        'Banana smoothie': {
            'prep time': 10,
            'nutrition per serving': ['Calories 122', 'Protein 5 g']
        }
    }
    test_cookbook(cookbook, 'Salmon salad', 'directions')
    test_cookbook(cookbook, 'Fried eggs', 'ingredients')
    test_cookbook(cookbook, 'Salmon salad', 'nutrition per serving')
    test_cookbook(cookbook, 'Apple pie', 'prep time')
    # empty cookbook dict
    test_cookbook(dict(), 'Tiramisu', 'ingredients')
    # other data types as keys
    cookbook = {15: {('1', '0'): 'ten', 'a': 'something'}}
    test_cookbook(cookbook, 15, ('1', '0'))
    test_cookbook(cookbook, 7, 'a')
```

CONSOLE VIEW

```
>python3 s02t01_dict_main.py
Searching for section "directions" in recipe "Salmon salad":
Cut tomatoes. Mix all ingredients together.
---
Searching for section "ingredients" in recipe "Fried eggs":
['eggs', 'salt', 'black pepper']
---
Searching for section "nutrition per serving" in recipe "Salmon salad":
There is no section "nutrition per serving" in the "Salmon salad" recipe.
---
Searching for section "prep time" in recipe "Apple pie":
There is no "Apple pie" recipe in the cookbook.
---
Searching for section "ingredients" in recipe "Tiramisu":
There is no "Tiramisu" recipe in the cookbook.
---
Searching for section "('1', '0')" in recipe "15":
ten
---
Searching for section "a" in recipe "7":
There is no "7" recipe in the cookbook.
---
>
```

SEE ALSO

Python Dictionary and Dictionary Methods
Python Dictionary Get() Method

Act Basic: Task 02

NAME

Set

DIRECTORY

`t02_set/`

SUBMIT

`witch_hunt.py`

BEFORE YOU BEGIN

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

- What is a `set` in Python?
- In what ways can you define a new `set`?
- Why can't a `set` contain a `list` as an element?
- What are the advantages of sets?
- What interesting methods do sets have?

Sets have a lot of useful methods, below is an example of how to use some of them. Open the PYTHON INTERPRETER and repeat the actions.

```
>python3
>>> # define several sets
>>> fruits = {'apple', 'pear', 'banana', 'orange', 'lime'}
>>> vegetables = {'carrot', 'cucumber', 'potato', 'garlic'}
>>> orange_foods = {'juice', 'carrot', 'cake', 'orange', 'apple'}
>>> # let's find out which foods are BOTH fruits and orange_foods
>>> fruits & orange_foods
{'orange', 'apple'}
>>> # both vegetables and orange_foods
>>> vegetables & orange_foods
{'carrot'}
>>> # the operation above is called intersection and can also be used as a method:
>>> vegetables.intersection(orange_foods)
{'carrot'}
>>> # now let's use the set difference operator to find fruits that are NOT orange_foods
>>> fruits - orange_foods
{'lime', 'banana', 'pear'}
>>> # using the set difference operator to find orange foods that are not fruits
>>> orange_foods - fruits
{'carrot', 'cake', 'juice'}
>>> # same but using method `difference()` instead of operator:
>>> orange_foods.difference(fruits)
{'carrot', 'cake', 'juice'}
>>>
```

DESCRIPTION

Create a script with a function `witch_hunt()` that looks for witches using sets of names. The function takes two lists of sets as arguments: `suspect_sets` and `innocent_sets`, and returns a set of names of witches. The logic is: a person, whose name is in every set of `suspect_sets` and is not in any set of `innocent_sets`, is a witch. So, if the `suspect_sets` are
`['a', 'b', 'c', 'd', 'd', 'b', 'a', 'e']`;
and the `innocent_sets` are `['d', 'f', 'g', 'x', 'y']`, then the returned set of witches must be `'b', 'a'`. You can see this example in the PYTHON INTERPRETER section.

Use the methods `intersection()` and `difference()` (or operators) to solve this task.

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

EXAMPLE

```
from witch_hunt import witch_hunt


def run_test(suspect_sets, innocent_sets):
    witches = witch_hunt(suspect_sets, innocent_sets)
    assert isinstance(witches, set)
    print(f'Witches found: {sorted(list(witches))}')


if __name__ == '__main__':
    can_read = {'Keeva', 'Daegal', 'Adam', 'Bellatrix', 'Ulrich', 'Keene',
                'Evanora', 'Earthan', 'Paxton', 'Alice', 'Candice', 'Cedonia',
                'Zelig', 'Lydia', 'Mortimer'}
    talk_to_self = {'Candice', 'Lydia', 'Chilton', 'Alice', 'Cedonia',
                    'Minerva', 'Adam', 'Daegal', 'Camilla', 'Keene',
                    'Chalmers', 'Keeva', 'Paxton'}
    healers = {'Bellatrix', 'Cullen', 'Adam', 'Alice', 'Lydia', 'Ulrich',
               'Zelig', 'Cedonia', 'Paris', 'Chalmers', 'Chilton',
               'Minerva', 'Paxton', 'Mortimer', 'Earthan', 'Daegal'}
    wealthy = {'Chalmers', 'Keeva', 'Alice', 'Cullen', 'Minerva'}
    men = {'Keene', 'Zelig', 'Mortimer', 'Adam', 'Ulrich', 'Chalmers',
           'Paxton', 'Cullen', 'Chilton', 'Earthan', 'Daegal'}

    run_test([can_read, talk_to_self, healers], [wealthy, men])
    run_test([], [wealthy, men])
    run_test([can_read, talk_to_self, healers], [])
```

CONSOLE VIEW

```
>python3 s02t02_set_main.py
Witches found: ['Cedonia', 'Lydia']
Witches found: []
Witches found: ['Adam', 'Alice', 'Cedonia', 'Daegal', 'Lydia', 'Paxton']
>
```

PYTHON INTERPRETER

```
>python3
>>> from witch_hunt import witch_hunt
>>> witch_hunt([{'a', 'b', 'c', 'd'}, {'d', 'b', 'a', 'e'}], [{'d', 'f', 'g'}, {'x', 'y'}])
{'b', 'a'}
>>> witch_hunt([],[])
set()
>>> witch_hunt([{10, 5, 6}, {7, 10, 2}],[])
{10}
>>> witch_hunt([],[{13, 4}, {15, 6, 0}])
set()
>>>
```

SEE ALSO

Python Sets: What, Why and How
Python Set intersection() Method
Python Set difference() Method

Act Basic: Task 03

NAME

Bubble Sort

DIRECTORY

`t03_bubble_sort/`

SUBMIT

`bubble_sort.py`

BEFORE YOU BEGIN

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

- What is sorting?
- What are the most popular sorting algorithms?
- What are the pros and cons of bubble sort?
- How does bubble sort work?

DESCRIPTION

Create a function `bubble_sort()` that sorts a list using the bubble sort algorithm. The elements in the list will only be of integer 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 bubble_sort.py, not the test script, and Oracle will check your function with random values.

EXAMPLE

```
from bubble_sort import bubble_sort


test_case_1 = [1, 2, 3, -1230]
test_case_2 = [3, 2, 1, 2, 2, 2, 3, 9, 8, 1, 3, 12, 32]


if __name__ == '__main__':
    print(f'Before sorting: {test_case_1}')
    bubble_sort(test_case_1)
    print(f'After sorting: {test_case_1}')

    print('***')
    print(f'Before sorting: {test_case_2}')
    bubble_sort(test_case_2)
    print(f'After sorting: {test_case_2}')
```

CONSOLE VIEW

```
>python3 s02t03_bubble_sort_main.py
Before sorting: [1, 2, 3, -1230]
After sorting: [-1230, 1, 2, 3]
***
Before sorting: [3, 2, 1, 2, 2, 2, 3, 9, 8, 1, 3, 12, 32]
After sorting: [1, 1, 2, 2, 2, 2, 3, 3, 3, 8, 9, 12, 32]
>
```

PYTHON INTERPRETER

```
>python3
>>> from bubble_sort import bubble_sort
>>> test_case_1 = [1, 2, 3, -1230]
>>> bubble_sort(test_case_1)
>>> test_case_1
[-1230, 1, 2, 3]
>>> test_case_2 = [3, 2, 1, 2, 2, 2, 3, 9, 8, 1, 3, 12, 32]
>>> bubble_sort(test_case_2)
>>> test_case_2
[1, 1, 2, 2, 2, 2, 3, 3, 3, 8, 9, 12, 32]
>>> test_case_3 = []
>>> bubble_sort(test_case_3)
>>> test_case_3
[]
>>> test_case_4 = [0, 1, -20, 0, 3]
>>> bubble_sort(test_case_4)
>>> test_case_4
[-20, 0, 0, 1, 3]
>>> test_case_5 = [0, 1, -20, 0, 3]
>>> test_case_5.sort()
>>> test_case_5
[-20, 0, 0, 1, 3]
>>>
```

SEE ALSO

Sorting Algorithms
Comparison Sorting Algorithms

Act Basic: Task 04

NAME

Unique

DIRECTORY

`t04_unique/`

SUBMIT

`unique.py`

BEFORE YOU BEGIN

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

- How to create a dictionary of lists?
- How to iterate over the values of a dictionary?
- How to get unique values from a list?
- How can the `set()` function help you get a unique list?

Below is a simple example of using `set()`.

```
>python3
>>> list_example = [1, 2, 1, 1, 4, 5]
>>> list(set(list_example))
[1, 2, 4, 5]
>>>
```

DESCRIPTION

Create a function `make_unique()` that:

- takes a dictionary with lists as values
- removes all duplicate items in every list
- sorts the elements of each list in the given dictionary in an ascending order
- returns the formatted dictionary

EXAMPLE

```
from unique import make_unique


test_case_1 = {
  "city": ['Kharkiv', 'Kyiv', 'Lviv', 'Dnipro', 'Kyiv', 'Kyiv', 'Kharkiv',
           'Kharkiv', 'Lviv'],
  "age": [16, 16, 17, 18, 19, 19, 19, 18, 20],
}
test_case_2 = {
  "hobby": ['drawing', 'basketball', 'drawing', 'drawing', 'basketball',
```

```
            'dancing', 'dancing', 'dancing', 'dancing'],
  "format": ['individually', 'individually', 'group', 'individually', 'group',
             'individually', 'group', 'group', 'group'],
}


if __name__ == '__main__':
    print('-----Before-----')
    print(test_case_1)
    print('-----After-----')
    print(make_unique(test_case_1))

    print('***')
    print('-----Before-----')
    print(test_case_2)
    print('-----After-----')
    print(make_unique(test_case_2))
```

CONSOLE VIEW

```
>python3 s02t04_unique_main.py
-----Before-----
{'city': ['Kharkiv', 'Kyiv', 'Lviv', 'Dnipro', 'Kyiv', 'Kyiv', 'Kharkiv', 'Kharkiv',
↪  'Lviv'], 'age': [16, 16, 17, 18, 19, 19, 19, 18, 20]}
-----After-----
{'city': ['Dnipro', 'Kharkiv', 'Kyiv', 'Lviv'], 'age': [16, 17, 18, 19, 20]}
***
-----Before-----
{'hobby': ['drawing', 'basketball', 'drawing', 'drawing', 'basketball', 'dancing',
↪  'dancing', 'dancing', 'dancing'], 'format': ['individually', 'individually',
↪  'group', 'individually', 'group', 'individually', 'group', 'group', 'group']}
-----After-----
{'hobby': ['basketball', 'dancing', 'drawing'], 'format': ['group', 'individually']}
>
```

PYTHON INTERPRETER

```
>python3
>>> >>> from unique import make_unique
>>> test_case_1 = {
...     "city": ['Kharkiv', 'Kyiv', 'Lviv', 'Dnipro', 'Kyiv', 'Kyiv', 'Kharkiv',
↪  'Kharkiv', 'Lviv'],
...     "age": [16, 16, 17, 18, 19, 19, 19, 18, 20],
... }
>>> print(make_unique(test_case_1))
{'city': ['Dnipro', 'Kharkiv', 'Kyiv', 'Lviv'], 'age': [16, 17, 18, 19, 20]}
```

```
>>> test_case_2 = {
...     "hobby": ['drawing', 'basketball', 'drawing', 'drawing', 'basketball', 'dancing',
↪  'dancing', 'dancing', 'dancing'],
...     "format": ['individually', 'individually', 'group', 'individually', 'group',
↪  'individually', 'group', 'group', 'group'],
... }
>>> print(make_unique(test_case_2))
{'hobby': ['basketball', 'dancing', 'drawing'], 'format': ['group', 'individually']}
>>>
```

SEE ALSO

Get Unique Values From a List in Python

Act Basic: Task 05

NAME

RegEx

DIRECTORY

`t05_regex/`

SUBMIT

`regex.py`

BEFORE YOU BEGIN

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

- What is a regular expression in Python?
- How to write regexes in Python?
- What module do you need to work with regular expressions?
- What functions does the module contain to search for a match in a string? How is their work different?
- What are metacharacters?
- What is character escaping? Where can you need it?
- What is the difference between `+` and `*` in regex?
- What are anchors in regex? And what are the anchors for the start and end of a string?

DESCRIPTION

Create a function `check_address()` that:

- takes a dictionary that contains the person's first and last name as key, and address as value
- checks every address on correct writing using a regular expression. Below you can find the rules
- returns a list of strings in the format `"The address of {key} is valid."` if the address is correct, otherwise `"The address of {key} is invalid."`

In this task you must use the `re` module.

The address must match the following format:
`Ukraine, [City], [Street name] [Building number], [Index]`:

- all info must be in the specified sequence
- `'Ukraine'` is case-sensitive
- `[City]` and `[Street name]` may contain letters, spaces, and hyphens, for example, Ivano-Frankivsk, New York, volodymyra velykoho
- both `[Building number]` and `[Index]` must contain only digits

- [Building number] must have between one and six characters
- [Index] must have exactly five characters
- there must be at least one space between [Street name] and [Building number]
- commas may be followed by any amount of spaces

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

The test script uses a function from the file s02t05_regex_random.py that can also be found in the resources. This file generates random values that can be a set of random characters, not real addresses. This is for validating the correctness of your regular expressions.

EXAMPLE

```
from regex import check_address
from s02t05_regex_random import run_random_tests


dictionary = {
        'Dmitry Chaykin': '   Ukraine ,  Kyiv  ,  Dorohozhytska    3,  04119 ',
        'Andrey Myskin': 'Ukraine, Lviv, volodymyra velykoho 52, 79053',
        'Tatyana Tsareva': 'Ukraine, Kyiv , Mykola Grinchenko 4, 03038',
        'Zhanna Akopyan': 'Ukraine, Kharkiv, 23 August 33, 61000',
        'Viktor Vasilyev': 'Ukraine, 5 Pasternaka Street Lviv 79000',
        'Andrey Sharov': '271 Akademika Pavlova Street, Kharkiv, Ukraine',
    }


if __name__ == '__main__':
    print(*check_address(dictionary), sep='\n')
    run_random_tests()
```

CONSOLE VIEW

```
>python3 s02t05_regex_main.py
The address of Dmitry Chaykin is invalid.
The address of Andrey Myskin is valid.
The address of Tatyana Tsareva is valid.
The address of Zhanna Akopyan is invalid.
The address of Viktor Vasilyev is invalid.
The address of Andrey Sharov is invalid.
>
```

PYTHON INTERPRETER

```
>python3
>>> from regex import check_address
>>> dictionary = {
...     'Dmitry Chaykin': '   Ukraine ,  Kyiv  ,  Dorohozhytska    3,  04119 ',
...     'Andrey Myskin': 'Ukraine, Lviv, Volodymyra Velykoho 52, 79053',
...     'Andrey Sharov': '271 Akademika Pavlova Street, Kharkiv, Ukraine'
... }
>>> check_address(dictionary)
['The address of Dmitry Chaykin is invalid.', 'The address of Andrey Myskin is valid.',
↪  'The address of Andrey Sharov is invalid.']
>>>
```

SEE ALSO

Python RegEx
Regular Expressions 101

Act Basic: Task 06

NAME

Generators

DIRECTORY

`t06_generators/`

SUBMIT

`cube.py`

BEFORE YOU BEGIN

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

- What are generators in Python?
- Why and when can they be used?
- What does the keyword `yield` mean in generators?

DESCRIPTION

Create a `cube()` generator function that takes an integer and returns the cube of the given number as long as the passed number is greater than zero.
The number decreases by one at each iteration.

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

EXAMPLE

```
from cube import cube


if __name__ == '__main__':
    n = 3
    print(f'***Passed parameter - {n}***')
    for i in cube(n):
        print(i)
    print('***')

    n = 8
    print(f'***Passed parameter - {n}***')
    for i in cube(n):
        print(i)
    print('***')

    n = 10
    print(f'***Passed parameter - {n}***')
    for i in cube(n):
```

```
        print(i)
    print('***')

    n = 0
    print(f'***Passed parameter - {n}***')
    for i in cube(n):
        print(i)
    print('***')

    n = -5
    print(f'***Passed parameter - {n}***')
    for i in cube(n):
        print(i)
    print('***')
```

CONSOLE VIEW

```
>python3 s02t06_generators_main.py
***Passed parameter - 3***
27
8
1
***
***Passed parameter - 8***
512
343
216
125
64
27
8
1
***
***Passed parameter - 10***
1000
729
512
343
216
125
64
27
8
1
***
***Passed parameter - 0***
***
***Passed parameter - -5***
```

```
***
>
```

PYTHON INTERPRETER

```
>python3
>>> from cube import cube
>>> cube(1)
<generator object cube at 0x10534d510>
>>> for i in cube(1):
...     print(i)
...
1
>>> for i in cube(5):
...     print(i)
...
125
64
27
8
1
>>> for i in cube(0):
...     print(i)
...
>>> for i in cube(-5):
...     print(i)
...
>>>
```

SEE ALSO

Generators
Python Generators

Act Basic: Task 07

NAME

Contacts

DIRECTORY

`t07_contacts/`

SUBMIT

`contacts.py`

BEFORE YOU BEGIN

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

- What dictionary method removes and returns an element by a given key?
- What is a 'word character' in the context of regex?

DESCRIPTION

In this task you will be working with a dictionary of email contacts. Each entry has an email as the key, and a dictionary of personal information as the value.

Create a script with a function `contacts()` that can:

- add a new entry to this dictionary of email contacts (or replace if exists)
- update an existing entry
- delete an existing entry

The function takes two dictionaries and a string, and returns a boolean.

The function `contacts()`:

1. takes the arguments:
   (a) `container`: a dictionary of email contacts (can be an empty dict)
   (b) `info`: a dictionary of personal information that must
      - contain an item with key - `'name'`, and value - a non-empty string with only word characters or spaces
      - contain an item with key - `'email'`, and value - a non-empty string that matches the pattern: `'[not @]@[not @].[not @]'`
   (c) `operation`: a string with the name of the operation
   Must be either `'add'`, `'update'`, or `'delete'`
2. returns `False` if either the `info` or the `operation` is not valid
3. performs an action corresponding to the given `operation` argument:
   - `'add'` adds an item to the `container` dict in the following way:
     - the key of the newly created item is the value of `'email'` in the `info` dict

- the value - the `info` dict, but without the `'email'` item
- if the `container` dict already has an item with such key, it gets overwritten

Returns `True`.

- `'update'` or `'delete'`
  Looks for a contact in the `container` with a key matching the email from the `info`. If the `container` does not have an item with this key, the function returns `False`.
  Otherwise:
  - `'update'` updates the found contact (the element in the `container` with key equal to the email value in the `info`) in the following way:
    * adds key-value pairs from `info` to the found contact if the key is not in the contact (note: don't add the `'email'` item in the `info`)
    * updates the values of the elements, the keys of which were already in the contact

    For example, updating a `container`:

    `'a@a.a': {'name': 'a', 'age': 25}, 'b@b.b': {'name': 'b', 'age': 37}`

    with an `info`:

    `'email': 'a@a.a', 'name': 'x', 'id': '5'`

    will result in the `container` becoming:

    `'a@a.a': {'name': 'x', 'age': 25, 'id': '5'}, 'b@b.b': {'name': 'b', 'age': 37}`

    Returns `True` after the update.
  - `'delete'` removes the found contact (the element in the `container` with key equal to the email value in the `info`) from the `container`. Returns `True`.

The script `s02t07_contacts_main.py` in the `resources` tests your function. It must produce output as seen in the `CONSOLE VIEW` section. Don't be alarmed at the length of the test script, it just generates some random values for you to work with. Use it as an example, add your own values, and also test your function as shown in the `PYTHON INTERPRETER`. Pay attention that you must only submit the file `contacts.py`, not the test script, and Oracle will check your function with random values.

CONSOLE VIEW

```
>python3 s02t07_contacts_main.py
___________add test___________
{
   ":1Y@x)t.o}Y": {
      "name": "EqV8i",
      "age": 69,
      "height": 193,
      "website": "8YtDt.com"
   },
   "hKc@4h\".rx^": {
      "name": "8X2Y4",
      "Lov": "Rbr"
   },
   ">Zb@I8Q.j2B": {
      "name": "IKpi9",
      "TLv": "JrG"
   },
   "j`5@HEK.hMM": {
      "name": "ezOLH",
      "website": "XnI3E.com"
   },
   "J^J@:|z.\t!2": {
      "name": "qNCIE",
      "website": "QC4vk.com",
      "vVP": "nKC",
      "height": 200
   }
}
_________update test__________
{
   ":1Y@x)t.o}Y": {
      "name": "V_O11",
      "age": 54,
      "height": 167,
      "website": "8YtDt.com",
      "JYq": "hDS"
   },
   "hKc@4h\".rx^": {
      "name": "8X2Y4",
      "Lov": "Rbr"
   },
   ">Zb@I8Q.j2B": {
      "name": "Czppv",
      "TLv": "JrG",
      "age": 96
   },
   "j`5@HEK.hMM": {
      "name": "ezOLH",
      "website": "XnI3E.com"
```

```
   },
   "J^J@:|z.\t!2": {
      "name": "na8sE",
      "website": "QC4vk.com",
      "vVP": "nKC",
      "height": 161
   }
}
_________delete test__________
_______validation test________
[False, False, False, False, False, False, True, False, False, True, False, False,
↪  False, False, False, True, False, True, True, True, False]
{
   ":1Y@x)t.o}Y": {
      "name": "V_O11",
      "age": 54,
      "height": 167,
      "website": "8YtDt.com",
      "JYq": "hDS"
   },
   "hKc@4h\".rx^": {
      "name": "8X2Y4",
      "Lov": "Rbr"
   },
   "T{\r@s!E.d5Q": {
      "name": "kRAiz",
      "website": "zuY_8.com",
      "SeI": "BnP",
      "age": 9
   },
   "[$G@p\rZ.HZI": {
      "name": "YKeq1"
   }
}
>
```

PYTHON INTERPRETER

```
>python3
>>> from contacts import contacts
>>> con, inf = {}, {'name': 'Bob', 'email': 'bob@gmail.com', 'job': 'builder'}
>>>
>>> ######################################## ADD ###
>>> contacts(con, inf, 'add')
True
>>> inf = {'name': 'Anne', 'email': 'anne@a.com', 'age': 25}
>>> contacts(con, inf, 'add')
True
>>> con
{'bob@gmail.com': {'name': 'Bob', 'job': 'builder'}, 'anne@a.com': {'name': 'Anne',
↪  'age': 25}}
>>> inf = {'name': 'Lisa', 'email': 'anne@a.com', 'pet': 'dog'}  # email already exists
>>> contacts(con, inf, 'add')
True
>>> con  # item was overwritten
{'bob@gmail.com': {'name': 'Bob', 'job': 'builder'}, 'anne@a.com': {'name': 'Lisa',
↪  'pet': 'dog'}}
>>>
>>>
>>> ######################################## UPDATE ###
>>> inf = {'name': 'Not There', 'email': 'not_present@in.container'}
>>> # since email not in container, there will be nothing to update
>>> contacts(con, inf, 'update')
False
>>> con  # no changes
{'bob@gmail.com': {'name': 'Bob', 'job': 'builder'}, 'anne@a.com': {'name': 'Lisa',
↪  'pet': 'dog'}}
>>> inf = {'name': 'Lisa', 'email': 'anne@a.com', 'pet': 'dog'}  # exists, no difference
>>> contacts(con, inf, 'update')
True
>>> con  # no changes
{'bob@gmail.com': {'name': 'Bob', 'job': 'builder'}, 'anne@a.com': {'name': 'Lisa',
↪  'pet': 'dog'}}
>>> inf = {'name': 'Lisa', 'email': 'anne@a.com', 'pet': 'cat'}  # exists, different pet
>>> contacts(con, inf, 'update')
True
>>> con  # value of 'pet' updated
{'bob@gmail.com': {'name': 'Bob', 'job': 'builder'}, 'anne@a.com': {'name': 'Lisa',
↪  'pet': 'cat'}}
>>> inf = {'name': 'Lisa', 'email': 'anne@a.com', 'age': 40}  # exists, no pet, new
↪  element
>>> contacts(con, inf, 'update')
True
>>> con  # added 'age' element
{'bob@gmail.com': {'name': 'Bob', 'job': 'builder'}, 'anne@a.com': {'name': 'Lisa',
↪  'pet': 'cat', 'age': 40}}
>>>
>>>
```

```
>>> ######################################## DELETE ###
>>> inf = {'name': 'Not even the correct name', 'email': 'anne@a.com'}  # exists
>>> contacts(con, inf, 'delete')
True
>>> con  # a contact was removed
{'bob@gmail.com': {'name': 'Bob', 'job': 'builder'}}
>>> inf = {'name': 'Bob', 'email': 'NOPE@gmail.com'}
>>> contacts(con, inf, 'delete')
False
>>> con  # no changes
{'bob@gmail.com': {'name': 'Bob', 'job': 'builder'}}
>>>
>>>
>>> ######################################## VALIDATION ###
>>> # invalid operation
>>> contacts(con, {'name': 'x', 'email': 'x@x.x'}, 'exterminate')
False
>>> # no 'name' or 'email' key in info
>>> contacts(con, {'name': 'x', 'not_email': 'x@x.x'}, 'add')
False
>>> # wrong pattern for 'name' or 'email'
>>> contacts(con, {'name': 'Name Okay', 'email': 'gmail_gone@.com'}, 'add')
False
>>> con  # no changes
{'bob@gmail.com': {'name': 'Bob', 'job': 'builder'}}
>>>
```

SEE ALSO

Regular Expressions Cheat Sheet

Act Advanced: Task 08

NAME

Contains

DIRECTORY

`t08_contains/`

SUBMIT

`contains.py`

DESCRIPTION

Create a function `contains()` that takes a string and a list of substrings.
The function checks if each substring is present in the string and returns a list of substrings that are in the string, otherwise it returns an empty list.
The function must ignore case.

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

EXAMPLE

```
from contains import contains


if __name__ == '__main__':
    # define the test cases
    str1 = 'Ah, I see you have the machine that goes \'ping\'. '\
           'This is my favorite.'
    substr1 = ['ping', 'machine']

    str2 = 'Strange women lying in ponds, distributing swords, is no basis '\
           'for a system of government!'
    substr2 = ['strange', 'no, basis']

    # find the present words
    res1 = contains(str1, substr1)
    res2 = contains(str2, substr2)

    # print the result
    print('-----Agruments-----')
    print(f"'{str1}'\n{substr1}")
    print('-----Detected-----')
    print(res1)

    print('***')

    print('-----Agruments-----')
```

```
    print(f"'{str2}'\n{substr2}")
    print('-----Detected-----')
    print(res2)
```

CONSOLE VIEW

```
>python3 s02t08_contains_main.py
-----Agruments-----
'Ah, I see you have the machine that goes 'ping'. This is my favorite.'
['ping', 'machine']
-----Detected-----
['ping', 'machine']
***
-----Agruments-----
'Strange women lying in ponds, distributing swords, is no basis for a system of
↪  government!'
['strange', 'no, basis']
-----Detected-----
['strange']
>
```

PYTHON INTERPRETER

```
>python3
>>> from contains import contains
>>> str1 = 'Ah, I see you have the machine that goes \'ping\'. This is my favorite.'
>>> substr1 = ['ping', 'machine']
>>> print(contains(str1, substr1))
['ping', 'machine']
>>> str2 = 'Strange women lying in ponds, distributing swords, is no basis for a system
↪  of government!'
>>> substr2 = ['strange', 'no, basis']
>>> print(contains(str2, substr2))
['strange']
>>>
```

Act Advanced: Task 09

NAME

Fibonacci

DIRECTORY

`t09_fibonacci/`

SUBMIT

`fib.py`

DESCRIPTION

Create a script that contains two functions:

- `fib()` takes a parameter `n` and returns the number in the Fibonacci Sequence under that index
- `fib_generator()` is a generator function. It takes a parameter (upper limit for the index) and returns the Fibonacci number under the current index. The function calls the function `fib()`

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

EXAMPLE

```
from fib import fib, fib_generator


if __name__ == '__main__':
    n = 20
    print(f'The number under the index {n} is {fib(20)}')
    print('***')

    n = 5
    print(f'The number under the index {n} is {fib(5)}')
    print('***')

    for i, n in enumerate(fib_generator(5)):
        print(f'Fibonacci number for {i} is {n}')
    print('***')

    for i, n in enumerate(fib_generator(10)):
        print(f'Fibonacci number for {i} is {n}')
    print('***')
```

CONSOLE VIEW

```
>python3 s02t09_fibonacci_main.py
The number under the index 20 is 6765
***
The number under the index 5 is 5
***
Fibonacci number for 0 is 0
Fibonacci number for 1 is 1
Fibonacci number for 2 is 1
Fibonacci number for 3 is 2
Fibonacci number for 4 is 3
***
Fibonacci number for 0 is 0
Fibonacci number for 1 is 1
Fibonacci number for 2 is 1
Fibonacci number for 3 is 2
Fibonacci number for 4 is 3
Fibonacci number for 5 is 5
Fibonacci number for 6 is 8
Fibonacci number for 7 is 13
Fibonacci number for 8 is 21
Fibonacci number for 9 is 34
***
>
```

PYTHON INTERPRETER

```
>python3
>>> from fib import fib, fib_generator
>>> fib(20)
6765
>>> fib(5)
5
>>> for i in fib_generator(5):
...     print(i)
...
0
1
1
2
3
>>> for i in fib_generator(10):
...     print(i)
...
0
1
1
2
3
```

```
5
8
13
21
34
>>> list((fib_generator(10)))
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
>>>
```

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.