Scanner Project Brief

Background

"My organisation has grown rapidly in a very short time and as a result the data management process has been rather inconsistent. There are a large number of documents scattered across various file shares and we need to ensure that we can locate any sensitive information to ensure it is not lost or exposed during a restructuring of our business documentation."

For this task you will be required to build a tool that can be used to find files which contain sensitive information such as:

- Classified Documents:
    - Documents which contain the term "Confidential", "Classified", etc.
    - These files might contain important company information which must not be lost or shared.
- Personal Content:
    - Documents which contain:
        - Phone numbers
        - Email addresses
        - Credit Card Numbers
        - Dates of Birth
        - National Insurance Numbers
- Potential Security Breaches:
    - Documents containing keywords indicating the presence of username or password information.

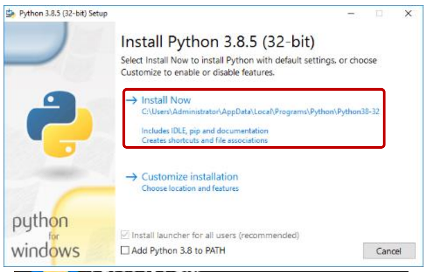
Broad Specification

1. The program must be able to traverse all the files in a given directory (folder). If a directory contains files and other sub-directories with files, the program must be able to systematically check all of them.
2. The program must reliably find matches in plain-text files based on pre-defined keywords and patterns (rules).
3. The program must be able to read in these rules from a configuration/rules file to allow the search to be customised by the end user.
4. The program must save its findings in a well formatted '.csv' file for easy viewing in spreadsheet software (eg. Microsoft Excel, OpenOffice Calc, Apple Numbers).

Introduction to This Project

For this project you will be using the Python 3 programming language. The latest releases of Python are available from https://www.python.org/

Getting Set Up



Python 3.8.5 (32-bit) Setup

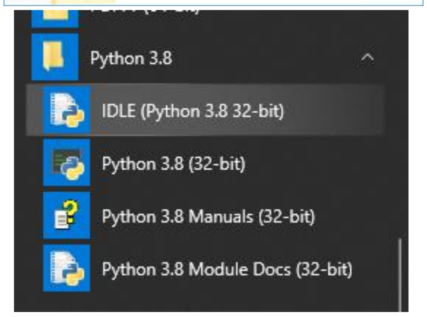
Install Python 3.8.5 (32-bit)

Select Install Now to install Python with default settings, or choose Customize to enable or disable features.

→ Install Now
C:\Users\Administrator\AppData\Local\Programs\Python\Python38-32
Includes IDLE, pip and documentation
Creates shortcuts and file associations
→ Customize installation
Choose location and features
☑ Install launcher for all users (recommended)
☐ Add Python 3.8 to PATH
Cancel
python for windows

When installing Python, it is recommended to use the default "Install Now" option. This will install Python, the included IDLE editor, and the Python launcher which is required for some of the tests to work.

Once installed you should be able to find the IDLE interpreter in your start menu or by searching for 'idle'.



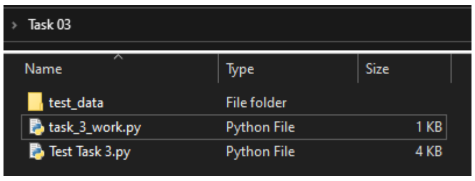
In IDLE you can run code manually (see Code Snippets below) and you can open new editor windows from the File menu. In the editor windows it is possible to run your code via the Run menu which will attempt run your code in the IDLE interpreter.

Code Snippets

The IDLE interpreter can be used to run Python code manually line-by-line to see the result and can be very useful for testing ideas. Throughout this project there are code snippets starting with '`>>>`', these can be copied into the IDLE interpreter and run for you to experiment with, you won't need to include the '`>>>`' when you copy the snippets.

Where to Put Your Work

There is an included work file for each task, and it is recommended that you use these files as your starting point as they are in the correct places and have the correct names. Some of them also contain templates and comments which might help.



› Task 03

Name	Type	Size
test_data	File folder	
task_3_work.py	Python File	1 KB
Test Task 3.py	Python File	4 KB

Testing Your Work

Each task has a test to check your work, it is important to make sure that your work is in the same folder as the test file and any test data, and that your work file is named correctly. You should be able to run the tests by simply double-clicking the "Test Task X.py" file. If the test is will not run or closes unexpectedly, you can try running the test file through windows command prompt; ensure the current working directory of the command prompt window is set to the location of the test file. Then run `py "Test Task X.py"` (Windows) or `python3 "Test Task X.py"` (Linux/OSX). This should run the test, and you will be able to see if something is going wrong.

Task 1: Basic Text and File Parsing

Your first task will be to write a function which can find keywords in a file, this will be the core of our project. We will do this in two steps, starting with text handling and then moving on to handling files.

Do all your work for this task in the file `task_1_work.py` in the Task 1 folder.

Part 1

Write a function called `get_hits` which takes two string parameters. The first parameter `text` will be a block of text to search, and the second parameter `keyword` will be a keyword to search for.

The function should get the count of how many times the keyword exists in the block of text. The value should be stored in a variable, and that variable should be returned.

To do:

- Create the `get_hits` function.
- Add your code to get the number of hits.
- Create and assign a variable.
- Return a variable.

Do not be surprised if the function looks very short at the end of this part.

Hint:

Strings in Python have a method called `.count` which can be used to easily get the number of times a piece of text appears inside another piece of text.

`my_string.count(test)` returns the number of times `test` appears in `my_string`:

```
>>> my_string = "Hello, World!"
>>> my_string.count("o")
2
>>> test = "Hello"
>>> my_string.count(test)
1
```

Part 2

Modify the `get_hits` function from Part 1 to accept a file-path instead of a block of text. The function should be modified to read in the text from the file, and check that instead.

Make sure the file is opened in ‘read’ mode.

To do:

- Modify the name of the `text` parameter to `path`.
- Add code to open a file and read its contents into a variable.
- Ensure your pre-existing code refers to the correct variable after your changes.

You might want to make some files of your own to read, and add some test code to the bottom of your Python file to test your code.

Check your code using ‘Test Task 1’

Task 2: Reading in Keywords

In this task we will tackle loading in keywords from a file. This is an important feature so that the search is customisable. Without doing this we would have to change our code every time we wanted to look for different things… not particularly useful.

Do all your work for this task in the file `task_2_work.py` which you can find in the Task 2 folder along with example keyword files.

Part 1

Write a function called `load_rules` which takes a single string parameter `path,` which will be the path of a file containing a single plaintext keyword (see `example_1.txt`).

The function must read in the keyword, remove any whitespace (spaces, new line characters, tabs, etc), and return the keyword.

To do:

- Create a new function called `load_rules`.
- Read the keyword from the file specified by the `path` parameter.
- Clean the keyword of any whitespace.
- Return the keyword.

Hint:
Python has a built-in method for removing characters from the beginning/end of a string. See the Python documentation here:
https://docs.python.org/3/library/stdtypes.html?highlight=strip#str.strip

Part 2

Modify your `load_rules` function to handle input files containing multiple keywords. The keyword files will have a single keyword on each line (see `example_2.txt`).

The function should return the keywords as a list of strings, and each keyword must be stripped of all whitespace/newline characters as in Part 1.

To do:

- Read in the file data as before.
- Split the data into lines.
- Strip all white space from each line leaving just the keywords.
- Ignore any blank lines/lines which only had white space characters.
- Return the list of keywords.

Hint:
Python has a built-in method for splitting a string based on where characters occur. See the Python documentation here:
https://docs.python.org/3/library/stdtypes.html?highlight=split#str.split

Check your code using ‘Test Task 2’

Task 3: Expanding ‘*`get_hits`*’

Our `get_hits` function from Task 1 can look for a single keyword in a file, but now that we can load multiple keywords from a rules file, it makes sense that we should expand our `get_hits` function accordingly.

Do all your work for this task in the file `task_3_work.py` in the Task 3 folder.

Copy your `get_hits` function created in Task 1 to `task_3_work.py`, and modify it to accept a list of keywords instead of a single keyword. Instead of returning a single count of hits, the function should return a list of counts which correspond to the given keywords.

To do:

- Change the `keyword` parameter to `keywords`.
- Create a new list variable to store your results.
- Add a `for` loop to repeat the `count` for each keyword, and store the results in the new list.
- Return your list of results.

Check your code using ‘Test Task 3’

Task 4: File Tree Traversal

Create a function called `scan`, which takes a string parameter `path` and prints the full relative paths of all the files within that directory, including any files in any sub-directories.

Do all your work for this task in the file `task_4_work.py` in the Task 4 folder.

Python's built-in '`os`' module has methods for traversing directories, specifically `walk(`*`path`*`)`, and `join(`*`path_part_1, path_part_2`*`)`.

The `walk` function can be used to get an object which can be iterated over in a `for` loop. On each iteration it will return a list in the format:
`[directory:` *`string`*`, sub-directories:` *`list of strings`*`, files:` *`list of strings`*`]`

The `join` function can take multiple parts of a file path and join them correctly, this is important as different operating systems handle file paths differently.
Its usage is as follows:

```
>>> from os.path import join
>>> join("C:\Users\alice\Desktop", "test file.txt")
"C:\Users\alice\Desktop\test file.txt"
```

The template file includes import statements for these two functions.

Expected Behaviour:

Folder Structure:

```
test_data
└───Structure 1
        │   File 1.txt
        │   File 2.txt
        │
        ├───Folder 1
        │       File 1.txt
        │       File 2.txt
        │
        ├───Folder 2
        │       File 1.txt
        │       File 2.txt
        │
        └───Folder 3
                File 1.txt
                File 2.txt
```

Parameter Values:

path: `test_data\Structure 1`

Printed Output:

```
test_data\Structure 1\File 1.txt
test_data\Structure 1\File 2.txt
test_data\Structure 1\Folder 1\File 1.txt
test_data\Structure 1\Folder 1\File 2.txt
test_data\Structure 1\Folder 2\File 1.txt
test_data\Structure 1\Folder 2\File 2.txt
test_data\Structure 1\Folder 3\File 1.txt
test_data\Structure 1\Folder 3\File 2.txt
```

Check your code using 'Test Task 4'

Task 5: Scanning a Directory

Part 1

In this task you will modify the code from Task 4 and use your `get_hits` function from Task 3. Copy your `scan` (Task 4) and `get_hits` (Task 3) functions into `task_5_work.py` and do all your work here.

After Task 4 your `scan` function should take a single parameter `path` and contain a call to `print` which occurs once for each file. For this task you will need to add a new parameter to your `scan` function once you've copied it. The parameter should be called `keywords` and will be a list of strings.

Next, replace the print statement with a call to your `get_hits` function. You will need to pass through values for its `path` and `keywords` parameters and store the result it returns in a variable.

Now we need a data structure to store our results. Dictionaries in Python store data as key-value pairs; the *value* being any kind of variable, and the *key* being some form of basic data such as a string or integer.

Values can be assigned to keys in the dictionary as follows:

```
>>> some_variable = ["This", "is", "a", "list"]
>>> my_dict = {}  # Creates a new empty dictionary.
>>> my_dict["key 1"] = some_variable
>>> my_dict
{'key 1': ['This', 'is', 'a', 'list']}
```

And a value can be retrieved as follows:

```
>>> retrieved = my_dict["key 1"]
>>> retrieved
['This', 'is', 'a', 'list']
```

Use a dictionary to store your results, where the *value* is the result returned by `get_hits`, and the *key* is the file path to which the result relates, as shown below:
`{'path 1': [int, int, …], 'path 2': [int, int, …], …}`
This 'results' dictionary will need to be created before you process any files, and then added to.

Finally, add a `return` statement to the end of your `scan` function so it returns the results when called.

Part 2

Add a check to ensure a result is only added to the results dictionary if there is at least one hit on the file which was checked. If there were no hits, the list of numbers returned by `get_hits` will be all zeros. So, you will need to check for at least one non-zero in the returned number and use an `if` statement to add the result if this is the case.

Check your code using 'Test Task 5'

Task 6: Sorting

In the next task we will be formatting and saving our results; part of formatting our results will be to sort them, so we will tackle the sorting first. In the next task, the data which we will be handling will be a list where each entry in the list will be another smaller list. You can think of this like storing a table of data, where each entry in the main list is a row, and each row is a list of values. This format, a list of lists, is often referred to a 2-dimensional (2D) list.

Write a function called `sort_2d_list` which takes three parameters; `data`: list, `index`: integer, `reverse`: boolean. The function should take a 2D list of data and sort it based on the values at the position of `index` in each entry. The function should sort the values from lowest to highest unless `reverse` is `True`, in which case it should sort highest to lowest. You should research a sorting algorithm such as “insertion sort” or “bubble sort” and create your own implementation.

Do your work in `task_6_work.py`, which contains a template to work from.

Expected Behaviour

List before sort:
`[["Charlie", 12], ["Alice", 32], ["Bob", 12], ["Daisy", 6]]`
List after sorting by index `0`:
`[['Alice', 32], ['Bob', 12], ['Charlie', 12], ['Daisy', 6]]`
List after sorting by index `1`:
`[['Daisy', 6], ['Charlie', 12], ['Bob', 12], ['Alice', 32]]`
List after being sorted by index `1` with `reverse=True`:
`[['Alice', 32], ['Charlie', 12], ['Bob', 12], ['Daisy', 6]]`

In Python it is possible to use the greater-than (`>`), less-than (`<`), and is-equal-to (`==`) comparisons to compare strings as well as simple numbers, as long as the two things being compared are of the same type[1]. A string will be greater than another if it would come after it alphabetically. For example, “`Banana`” is greater than “`Apple`”. This will work for longer strings as well, not just single words. If the two strings are largely the same Python will go through the strings and find the first point at which they can be ordered. So, it will say that “`aaaaaaaaaab`” is greater than “`aaaaaaaaaaa`”.

It is important to note that not all languages are so helpful when comparing strings, but most languages have some way of making this comparison.

The function should return the list after it has sorted it.

Check your code using ‘Test Task 6’

[1] In some cases, Python will support `<` or `>` comparisons between different types. A classic example is when comparing `int`’s and `float`’s; in this case Python will properly compare the two numbers without issue. If you try and compare a `string` and `int` with `<` or `>` you will get a `TypeError`. The area of equality testing and comparisons in Python is complex and adaptable.

Task 7: Formatting and Saving

Part 1

For this task you will need to create a new function called `format_results` which takes two parameters, `results` and `keywords`.

Do all your work for this task in `task_7_work.py`.

The `results` parameter should accept a dictionary of results which will be in the same format as we created in the Task 5, and `keywords` will be the same list of string keywords as we have been using previously.

Your new function should create and return a 2D list, where each entry is a list of values which will form a row of an output `.csv` file. A single file may have hits for multiple keywords. If so, you will need to create multiple rows in the output for that file. You should also *not* create a new row for a keyword if the number of hits for it is 0.
See the example output below, where “keyword 2” has no entry for “path 1” because it had 0 hits, likewise for “keyword 3” and “path 2”.

Once all the rows have been created, they should be ordered (in decreasing order of priority) by filename alphabetically, then by the number of hits with the highest to lowest, and then by keyword alphabetically. Copy in and use your `sort_2d_list` function from Task 6 to do this. You will need to perform multiple successive sorts.

Expected Behaviour:

Input:

```
results
{
  "path 1": [1, 0, 4],
  "path 2": [2, 1, 0]
}
keywords
(
  "keyword 1",
  "keyword 2",
  "keyword 3"
)
```

Output:

```
[
  ["path 1", "keyword 3", 4],
  ["path 1", "keyword 1", 1],
  ["path 2", "keyword 1", 2],
  ["Path 2", "keyword 2", 1]
]
```

Hint:

When you want to sort a list by multiple criteria in decreasing order of priority; in this case file name, then hits, then keyword; your first thought might be to perform your sorts in that order. But you’ll likely find that this doesn’t work. Why doesn’t this work? In what order can you perform your sorts such that it fixes the problem?

Part 2

Now you must write a function called `save_results` which takes two parameters; an output file path as `output_path`, and the 2D list of data to be saved as `results`.

The function must create a new file with the name given in `output_path`, and write the data from `results` into the file in comma separated values (CSV) format.

You will need to write the header line to the file before writing any data lines.

Expected Behaviour:

Input:

```
output_path: "output.csv"
results:
[
  ["path 1", "keyword 2", 4],
  ["path 1", "keyword 1", 1],
  ["path 2", "keyword 1", 2],
  ["Path 2", "keyword 2", 1]
]
```

Output:

`output.csv`

```
File Path,Keyword,Hits
"path 1","keyword 2",4
"path 1","keyword 1",1
"path 2","keyword 1",2
"path 2","keyword 2",0
```

Notice that in the example output above (right) that the values for File Path and Keyword have been given quote marks (") before and after. These are characters which are written to the file, not just markers of a string variable as they are in the input (left). They are not always a requirement for CSV but are here to prevent errors. Do some reading into CSV formatting to understand what kind of errors they will help prevent.

For your solution to pass you should include quote marks around the first two values in each row.

Note:
When dealing with files, it is possible for things to go wrong. For example; the file may already exist, the program may not have permission to write to the file, or one of many other possible problems depending on the situation.
As such it is good practice to prepare for these possible failures and include code for when things go wrong. This is called 'exception handling' and is not something we will be covering in this project, but it is a topic worth reading into if you intend to make versatile projects which are resistant to failure.

Check your code for Parts 1 & 2 using 'Test Task 7'

Task 8: Putting it All Together

In this task, the project will come together into the Minimum Viable Product (MVP). Do all your work in the `task_8_work.py` file, and bring in your previous work. Copy in the following functions:

- `load_rules` from Task 2
- `get_hits` from Task 5
- `scan` from Task 5
- `sort_2d_list` from Task 7
- `format_results` from Task 7
- `save_results` from Task 7

Don't forget, you will also need to include any `import` statements at the top of your work which have been used in Tasks 2, 5, or 7.

You will now need to make a new function called `main`. Your new `main` function will not need to take any parameters. It will handle the user input and calling of necessary functions to complete a scan.

The `main` function should ask for user input with the following prompts:

- `"Enter directory to search: "`
- `"Enter keywords file path: "`
- `"Enter output file path: "`

You will need to use the built-in function `input(`*`prompt`*`)` to get the user input. It will display a given prompt and wait for the user to enter a response, which it will return as a string when they press Enter:

```
>>> name = input("What is your name?: ")
What is your name?: Alice
>>> print("Hello " + name + "!")
Hello Alice!
```

Using the `input` function, ask for the user's inputs and store them in appropriately named variables. Then make calls to the functions which you have copied in to complete a scan and save the results.

After scanning and saving, the `main` function should show the message:
`"Scan complete! Press <ENTER> to close..."`
and wait for the user to press Enter before finishing. You can do this by using a `print` and an `input` call with an empty prompt; or just use the `input` function with this as the prompt.

The test will simulate a user using your program, and so you *must* use the prompts as shown above exactly and in the same order, or your program will not pass the tests. Before testing, ensure that the only code in the `__name__ == "__main__"` section is a call to your new `main` function.

Check your code using 'Test Task 8'

Task 9: Introducing Regular Expressions

In this task you will begin looking at regular expressions, or regex for short. Regular expressions can be used in computing to quickly find patterns in text; the patterns can be as simple as a single word, or extremely complex and able to catch very specific patterns.

For this task you will need to make a basic regular expression to pick out potential dates in a piece of text. Create a regular expression to match a string of the form "`dd/dd/dd`" or "`dd/dd/dddd`", where "`d`" represents a decimal digit.

You can use the website https://regexr.com/ to help you.

Check your pattern using 'Test Task 9'

Afterthought:

How might this approach cause problems in its current form?
What other information might this catch other than dates?
How might this pattern fail to catch some dates?

Could this regex accidentally match something like a product code if it looked like 36/11/1231? Some dates use '`.`' or '`-`' to separate their days, months, and years, does this catch them?

Task 10: Expanding ‘`get_hits`’ Again

Copy all your existing code from Task 8 into `task_10_work.py`, and make all modification for this task in the Task 10 file.

In this task, you will modify your `get_hits` function once again. Python 3 includes a library called “`re`” which provides regular expression functionality.
It contains multiple functions for finding matches, the most useful for our purposes will be `findall(`*`pattern, string`*`)`. The `findall` method will take a pattern and some text, and return a list of the matches it finds, if it finds no matches then the list will be empty.

See the documentation here for more information:
https://docs.python.org/3/library/re.html?highlight=re%20findall#re.findall

Handily, the regex pattern for matching a single keyword will simply be the keyword itself. As such we can modify `get_hits` to treat the keyword as a regex pattern instead, and it will behave the same as before for keywords whilst also increasing its functionality by accepting regex patterns.

Python also includes a built-in function called `len(…)` which can be used to get the length of many different types of data including lists, and so will be useful for getting the number of hits by getting the length of the list returned by the `findall` function.

Import `findall` from the `re` library by adding the line “`from re import findall`“ at the top of your program, then use `findall` and `len` to replace your usage of `.count` in your `get_hits` function using the keyword as the pattern.

Check your code using ‘Test Task 10’

Afterthought:
If we want to expand this product in the future, just having a count of matches might not be particularly helpful. What other information about matches might be useful to find, and how might that be done?

One option is to record where in the text the match was found.
The re module has a „search" method which return the first match it finds including where it was found. When you call „search" you can specify where it searches from, making it possible to work your way through the text find each new match and know where each match occurred.

Task 11: Challenges

You should now have a working program capable of finding patterns and keywords in plain-text files, so now it’s time to put it to good use!

In the Task 11 folder you will find a large collection of files in a .zip file. Extract this zip file and copy in your completed scanner from Task 10. In the below challenges you will need to find certain kinds of files from this collection. The dataset contains over 1000 files, so your program will be an valuable tool in finding the answers.

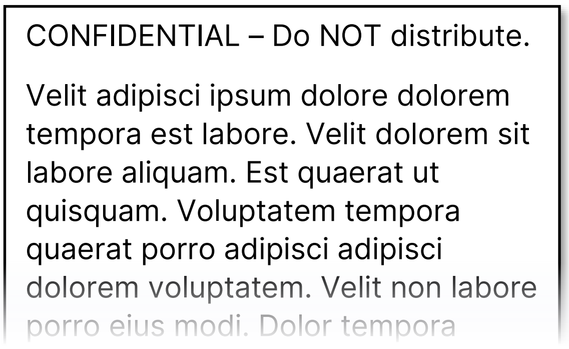
Each file has a unique 4-character name; check answers for each challenge by running the `Answer Checker.py` script and entering the 4-character codes of the files you found as one long string. For example, if you found these files for challenge 1;
`folder1\AB12.txt` and `folder2\folder3\CD34.txt`
you would enter your answer as ‘AB12 CD34’, don’t worry about the order of the files.

You will need to be cautious. There may be files which will catch you out with data that is close to, but not quite, what you are looking for. You will likely need to check your results manually, but well written regex patterns will help reduce the number of files you need to check.

Challenge 1:

One way that important documents are marked is with a header at/near the top of the file which says that the file is confidential, private, secret, etc.

For this challenge you will need to find the 12 files hidden in the dataset which have the header ‘CONFIDENTIAL’, ‘SECRET’, or ‘PRIVATE’. These headers may be upper case, lower case, or sentence case.
E.g. ‘SECRET’, ‘secret’, and ‘Secret’.



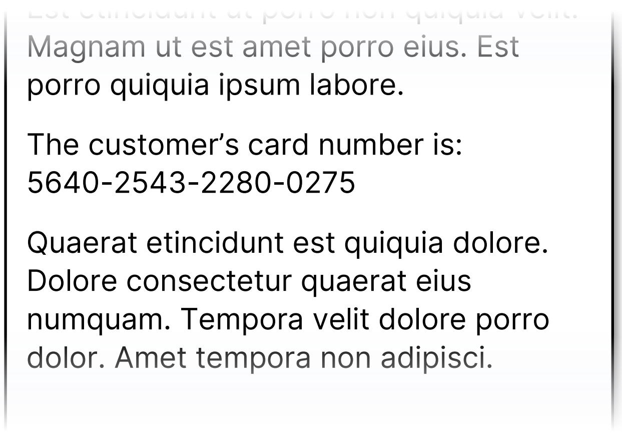
CONFIDENTIAL – Do NOT distribute.

Velit adipisci ipsum dolore dolorem tempora est labore. Velit dolorem sit labore aliquam. Est quaerat ut quisquam. Voluptatem tempora quaerat porro adipisci adipisci dolorem voluptatem. Velit non labore porro eius modi. Dolor tempora

Challenge 2:

Bank card numbers are personal data which must be protected, and corporate financial details could pose a serious risk if they end up in the wrong hands. For this challenge you must find the 10 files in the dataset which contain bank card numbers.

The bank card numbers will be 16 digits grouped into 4 groups of 4 digits, with either a space or hyphen used to separate the groups, *not a mix*.



Est etincidunt ut porro non quiquia velit. Magnam ut est amet porro eius. Est porro quiquia ipsum labore.

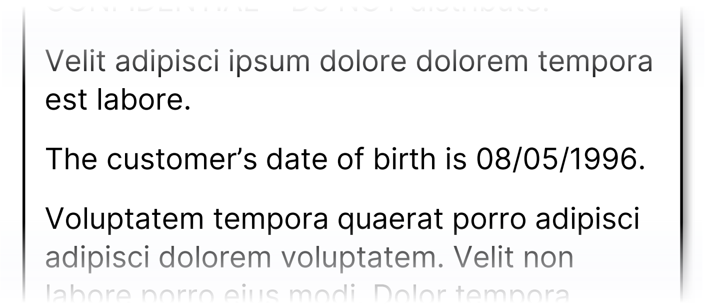
The customer’s card number is:
5640-2543-2280-0275

Quaerat etincidunt est quiquia dolore. Dolore consectetur quaerat eius numquam. Tempora velit dolore porro dolor. Amet tempora non adipisci.

Challenge 3:

Dates can be sensitive information, either as private data or in relation to business plans/strategies etc. You must find the 10 files in the dataset which contain dates. These dates will be of the format dd/mm/yy or dd/mm/yyyy.

You will need to be able to match both formats; there will not be any dates in other formats such as mm-dd-yyyy.



Velit adipisci ipsum dolore dolorem tempora est labore.

The customer's date of birth is 08/05/1996.

Voluptatem tempora quaerat porro adipisci adipisci dolorem voluptatem. Velit non labore porro eius modi. Dolor tempora

Hint:
To help reduce the number of results which you will need to check manually, you may wish to format your regex pattern(s) to catch day values only in the range 1-31, and month values only in the range 1-12.

Afterthought:
Whilst hunting for your answers you likely got a lot of false positives giving you more files you had to look through by hand to find the right ones. So, how could we reduce the number of false positives?

In many cases the best solution is to look at the wider context of the data; where was the match in the file, what else was it close to, and does the matched text meet other specific criteria such as check sums and formatting. In the next task we will look at one approach which can help with this.

Task 12: An Introduction to Validators

No matter how well constructed your regex patterns are, there is still the possibility of false positives. For example, when looking for dates in the last task, you may have caught 31/02/1970, which is not a valid date as February never has more than 29 days. Checking the validity of dates might not be practical to implement in regex.

Similarly, there are cases where 16 digits might occur together other than as a bank card number; serial numbers, product codes, software keys etc. Luckily, bank card numbers include a check digit which can be used to check that the number is valid, and we can use it to increase our confidence of a valid match.

A validator is a small dedicated function, designed to check a particular type of data and tell us whether or not it fits the format that the validator is built for. When handling data which we suspect to be a match, we can use a validator to check that it is a solid match.

Do your work for this task in `task_12_work.py`.

Part 1: Luhn 10

The Luhn 10 algorithm is a way of generating a check digit from a given series of numbers and is used in various applications including UK bank card numbers. The Luhn 10 process can then be used to check a number with a check digit to ensure it is valid. This is because any changes/errors in the number will cause the check digit to no longer match.

Write a function called `validate_luhn10` which takes a string of digits as a parameter (not an integer as this would lose leading zeros). It should return `True` or `False` depending on whether the string passes the Luhn 10 check. You will need to do your own research into how the Luhn 10 check is performed and implement it.

Check your code using 'Test Luhn 10'

Part 2: Dates

Validating dates may seem simple at first sight, however, this gets more complex when you start to account for leap years as well as the number of days allowable for certain months.

Write a function called `validate_date` which takes a string representation of a date in dd/mm/yyyy format and returns `True` or `False` depending on whether the string passes as a valid date.

Hint
You may want to start by writing a function for validating whether a year is a leap year. You can then use this function to help you in your date validator.

Check your code using 'Test Date'