CSC110 Fall 2022 Assignment 1: Data and Functions

Print this handout

In this assignment, you will apply what you've learned so far in this course to a variety of different problems. In Parts 1 and 2 you'll complete some short exercises that check your understanding of data types, expressions, and functions in Python. In Part 3, you'll practice *debugging* to identify errors in a small Python program. And finally in Parts 4 and 5, you'll apply what you've learned so far in CSC110 t a larger task of performing simple image manipulations like cropping parts of an image and applying specific colour filters.

We hope you enjoy working on this assignment, and feel a sense of accomplishment as you work through each part!

Advice for Approaching Assignment 1

This is your first assignment for CSC110, and so we've prepared some advice to help you navigate and complete this assignment early on in the semester.

First impressions

After we've posted a new assignment, you can always do the following three prep tasks early on, and they shouldn't take too much time.

1. **Skim the entire assignment handout once.** You don't need to go into detail or feel like you understand every single part. Just make sure you know what all of the different parts of the assignment are, and start making connections to what you're learning in lecture or practicing in tutorial.

2. **Download all of the starter files from MarkUs.** There are instructions on how to obtain the starter files for the assignment below, and again you should do this very early on so that you have all of the necessary files to complete the assignment.

3. **Schedule time to work on the assignment (put it in your calendar!).** You'll typically have around two weeks to work on each assignment. Even if you aren't ready to start an assignment the day it comes out, you can plan out your next two weeks to find time to work on the assignment.

   Warning: we don't recommend trying to complete assignments in one sitting. You'll want to make sure you take breaks so that you don't burn yourself out, and often you might get stuck on something and want to return to it in a few days, perhaps after having asked questions on Campuswire or during office hours.

Breaking the assignment down

Assignments are the biggest type of assessments you'll complete in CSC110, and their scale can feel a bit intimidating at first. To help you, we typically break assignments down into around 4-5 parts, where each part consists of a number of questions/problems to solve. Importantly, *each assignment part is often independent of the other parts*, meaning even if you get stuck on one part you can usually move onto the next part and come back later. (This is mostly true for Assignment 1, except Part 5 will build on your work in Part 4.) Individual questions within each part may or may not be independent, though usually each part will have a common "theme" or problem domain that relates the various questions.

As you schedule time to work on the assignment, you might find it useful to set a deadline for each part separately. For example, for Assignment 1 you might say:

- I'll finish Parts 1 and 2 by Sunday
- I'll finish Part 3 by next Wednesday
- I'll finish Parts 4 and 5 (the biggest parts) by the following Monday.
- I'll spend the last two days reviewing and improving my work before making my final submission.

Connecting assignment parts to what you've learned

Assignments are meant to evaluate not just the basic knowledge and skills you've learned, but also your ability to apply your learning to new domains, and to synthesize multiple concepts and skills to solve more complex problems. Because of this, assignments aren't just going to repeat what you've done in lecture or tutorial!

However, assignments will certainly be *connected* to what you've learned, and it is useful when approaching an assignment to remind yourself about exactly what those connections are. This can help you complete and assignment, but also figure out what you need to go back to review if you get stuck.

For Assignment 1, here are some explicit connections:

- **Part 1** consists of short answer questions connected to data types and comprehensions, which is covered in Chapter 1 of the Course Notes (and in the first two lectures).
- **Part 2** contains some programming exercises that are similar in structure to the functions defined in Week 1's lectures and tutorial, and Prep 2.
    - We're releasing Prep 2 at the same time as Assignment 1, and we recommend working through the programming exercises on the prep before working on this part of the assignment.
- **Part 3** is a code *reading* exercise which is similar to the "explain why an error occured" exercises in Week 1's lectures and tutorial, though at a more sophisticated level.
    - Part 3 also makes use of pytest, which will be covered at the start of Week 2. So, you might consider waiting to work on it until after that lecture, or read ahead to Section 2.8 (https://www.teach.cs.toronto.edu/~csc110y/fall/notes/02-functions/08-testing-functions-1.html) in the Course Notes.
- **Parts 4 and 5** are application-heavy tasks with a focus on operating on colour values.
    - The Course Notes Section 1.8 (https://www.teach.cs.toronto.edu/~csc110y/fall/notes/01-working-with-data/08-representing-colour.html) provides a helpful overview of how we can represent colour values.
    - You'll be working with colour values in your first tutorial at the end of Week 1, so you might consider waiting to work on this part until after your tutorial.

> You'll notice that sometimes we'll release assignments where we haven't covered material connected to every part yet. We do this to allow you to begin working on the "first impressions" tasks early, and then get to work on at least some parts of the assignment!

Logistics

- Due date: Wednesday, September 28 before *12pm noon* Eastern Time.
  - We are using **grace tokens** to allow you to submit your work up to 30 hours after the due date. Please review the Policies and Guidelines: Assignments page (https://q.utoronto.ca/courses/278340/pages/policies-and-guidelines-assignments?module_item_id=3928858) carefully!
- This assignment must be done **individually**.
- You will submit your assignment solutions on MarkUs (see Submission instructions at the end of this handout).

Starter files

To obtain the starter files for this assignment:

1. Login to MarkUs (https://markus.teach.cs.toronto.edu/2022-09) and go to CSC110.
2. Under the list of assignments, click on **Assignment 1**.
3. On the Assignment 1 page, click on the **Download Starter Files** button (near the bottom of the page). This will download a zip file to your computer.
4. Extract the contents of this zip file into your `csc110/assignments/` folder.
5. You should see a new `a1` folder that contains the assignment's starter files!

You can then see these files in your PyCharm `csc110` project, and also upload the `a1.tex` file to Overleaf to complete your written work.

General instructions

This assignment contains a mixture of both written and programming questions. All of your written question work should be completed in the `a1.tex` starter file, using the LaTeX typesetting language. You went through the process of getting started with LaTeX in the Software Installation Guide (https://q.utoronto.ca/courses/160038/pages/setting-up-your-computer-start-here?module_item_id=1346385), but for a quick start we recommend using the online platform Overleaf (https://www.overleaf.com/) for LaTeX. Overleaf also provides many tutorials (https://www.overleaf.com/learn/latex/Learn_LaTeX_in_30_minutes) to help you get started with LaTeX.

> **Tip**: See this video tutorial (https://www.overleaf.com/learn/how-to/How_to_upload_files_to_a_new_project) for instructions on how to upload a `.tex` file to Overleaf.

Your programming work should be completed in the different starter files. Each part typically has its own starter file(s). We have provided code at the bottom of some of these files for running doctest examples and PythonTA on these two files. We'll discuss in lecture what this code does, and why it's useful for you!

> **Warning**: one of the purposes of this assignment is to evaluate your understanding and mastery of the concepts that we have covered so far. So on this assignment, you may only use the parts of the Python programming language that we have covered in Chapters 1 and 2 of the Course Notes. Using other parts of the Python language (e.g., loops, if statements) is not allowed, and the parts of your submissions that use them will receive a grade as low as **zero** for doing so.

Running PythonTA

We'll be using a program called *PythonTA* to check your work for common logical, design, and style errors, and this will determine a portion of your grade for all programming parts of this assignment.

As part of the Software Installation Guide (https://q.utoronto.ca/courses/160038/pages/setting-up-your-computer-start-here?module_item_id=1346385), you installed PythonTA on your own computer as well! This means that you can run PythonTA as you are working on your assignment to help identify

issues and so that you can fix them as they come up, and ensure that you'll get a perfect PythonTA grad on your final submission for this assignment. For instructions on how to run PythonTA, please check out our Assignment 1 Python Guide (using_python_ta.html).

Part 1: Data and Comprehensions

Write your responses in Part 1 of `a1.tex`, filling in the places marked with TODOs.

1. **Imagine this scenario...**

   > After saving money for an entire year, you and your sister are now planning a backpacking trip across Europe. You are responsible for planning where you'll be staying and making a plan for what to do on each day of your trip. Since you'll be gone for a long period of time, you are searching for hostels and AirBnB rentals that include laundry service. You also record the location of famous landmarks (like the Eiffel Tower) in each city you'll visit.
   >
   > Your sister is in charge of finding restaurants (she is a huge foodie!) and packing for the trip. She sends you her "top ten restaurants" with names and, of course, pictures of the food. She also writes down all items that you need to pack, and right before your leave for the airport, she weighs each suitcase to make sure their weights do not exceed the flight's luggage weight limit.

   For each of the following data, describe what Python data type would best represent this data, and briefly explain your answer in one or two sentences each. If the data type is a collection, include the type of its elements or key-value pairs in your response. *Note*: in at least some cases, there is more than one "correct" answer. Your TAs will be evaluating your justification for your choices.

   a. The total amount of money you have budgeted to spend on your trip.

   b. The restaurant names in your sister's "top ten restaurants" message, in the order of her preferences.

   c. The number of places you are staying that have laundry service.

   d. The names of the cities you will be visiting and the corresponding number of days you are staying in each city.

e. Whether or not you have a valid passport.

2. **Exploring comprehensions.** In class, we studied list, set, and dictionary comprehensions as useful tools in Python for building up large quantities of data. In this question, you'll demonstrate your understanding of comprehensions and how they can be used in new ways.

   a. Type the following into your Python console.

   ```
   >>> [c for c in 'Blueberry']
   ```

      i. What value does this expression evaluate to?

      ii. What is the value's data type, and what is the type of each of its elements?

   b. Now consider the expression `{c for c in 'Blueberry'}`.

      i. What does this expression evaluate to?

      ii. What is this value's data type, and what is the type of each of its elements?

      iii. Explain how this value differs from the value in part (a), in terms of their size and elements.

   c. Now, suppose we define the variable `names = ['David', 'Tom', 'Mario']`. Consider these two expressions:

   ```
   # Expression 1
   [name + '!' for name in names]

   # Expression 2
   [name for name in names] + ['!' for name in names]
   ```

   What does each expression evaluate to, and why are their values different?

Part 2: Programming Exercises

Open the starter file `a1_part2.py`, which contains a set of programming exercises. Follow the instructions found in the file to complete these exercises. *Note*: these instructions are the same as Prep 2's programming exercises.

You can test your work using the Python console (like Prep 2) or by using the `doctest` Python library, which we will cover in lecture and in the Course Notes Section 2.8 (https://www.teach.cs.toronto.edu/~csc110y/fall/notes/02-functions/08-testing-functions-1.html).

Part 3: Interpreting Test Results

Fun fact: David and Tom are secret owners of *Dancing Poutine*, a super cool restaurant and karaoke ba where customers can eat poutine and sing their favourite songs on stage.

In this restaurant, customers are charged:

- The menu price for their food and drink orders, plus 13% tax and 8% tip (the tip percentage is calculated using the menu price, not the after-tax amount).
- An additional $5 fee per song they sing (no tax is applied).

For example, if Angela visits the restaurant and buys one $10 poutine and sings nine songs, her total would be

$$10.0 + 10.0 \times 0.08 + 10.0 \times 0.13 + 9 \times 5.0 = 57.10$$

To encourage spending, David and Tom give a daily special prize to the customer that spends the most amount of money at their restaurant. To help them identify this customer quickly each night, they writ a small Python program contained in `a1_part3.py`. The main function in this file has the following header and docstring:

```
def get_largest_bill(bills: list) -> float:
    """Return the highest bill amount paid by a customer, rounded to two
        decimal places.

    Each element of bills is a dictionary with two key-value pairs:

    - key 'food', which corresponds to the total menu price of the food and
        drink
      that the customer ordered
    - key 'songs', which corresponds to the number of songs the customer sang

    You may ASSUME that:
        - bills is non-empty
        - every element of bills is a dictionary with the format described
        above
        - all menu totals and number of songs are >= 0
    """
```

David and Tom also writes **three** unit tests using `pytest` for this function. You can find the code for their program and tests in the starter file `a1_part3.py`.

Unfortunately, when they run `pytest` to test their work, they see the following output (some extraneo text has been removed):

```
======================================================== test session starts
        ========================================================

collected 3 items


a1_part3.py::test_single_bill PASSED                                 [ 33%]
a1_part3.py::test_two_customers FAILED                               [ 66%]
a1_part3.py::test_just_food FAILED                                   [100%]

============================================================== FAILURES
        ============================================================
_________________________________________________________ test_two_customers
        _________________________________________________________


    def test_two_customers() -> None:
        """Test get_largest_bill when there are two different bills.
        """
        bills = [
            {'Food': 15.0, 'Songs': 3},
            {'Food': 16.2, 'Songs': 2}
        ]

        expected = 33.15
>       actual = get_largest_bill(bills)

a1_part3.py:79:
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
        _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
a1_part3.py:41: in get_largest_bill
    return max([calculate_total_cost(bill['songs'], bill['food']) for bill in
        bills])
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
        _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

.0 = <list_iterator object at 0x000001FDD996B100>

>   return max([calculate_total_cost(bill['songs'], bill['food']) for bill in
        bills])
E   KeyError: 'songs'
```

```
a1_part3.py:41: KeyError
_____________________________________________________________ test_just_food
        ______________________________________________________________

    def test_just_food() -> None:
        """Test get_largest_bill when the customers only ordered food and
        drink, but no songs.
        """
        bills = [
            {'food': 10.0, 'songs': 0},
            {'food': 17.3, 'songs': 0},
            {'food': 21.25, 'songs': 0},
            {'food': 3.1, 'songs': 0},
            {'food': 0.0, 'songs': 0}]

        expected = 25.71
        actual = get_largest_bill(bills)
>       assert actual == expected
E       assert 106.25 == 25.71

a1_part3.py:95: AssertionError
================================================== short test summary
        info ===================================================
FAILED a1_part3.py::test_two_customers - KeyError: 'songs'
FAILED a1_part3.py::test_just_food - assert 106.25 == 25.71
================================================== 2 failed, 1 passed in
        0.59s ==================================================
```

Answer the following questions about this `pytest` report and the program found in `a1_part3.py`. Write your responses in `a1.tex`, except 2(b), where we ask you to edit `a1_part3.py` directly.

1. Looking at the `pytest` report, identify which tests, if any, passed, and which tests failed. You can just state the name of the tests and whether they passed/failed, and do not need to give explanations here.

2. For *each* failing test from the `pytest` report:

    a. Explain what error is causing the test to fail.

    Hint: the errors may occur in the function bodies, or in the tests themselves.

b. Edit `a1_part3.py` by fixing the function code and/or the tests so that all tests pass. The changes must be to fix errors only; the original purpose of the functions and tests must remain the same.

   Hint: Each failing test can be fixed with changing just one or two lines of code.

3. For *each* test that passed on the original code (before your changes in Question 2), explain why that test passed even though there were errors in the code.

Part 4: Colour Rows 🌈

In Section 1.8 of the Course Notes (https://www.teach.cs.toronto.edu/~csc110y/fall/notes/01-working-with-data/08-representing-colour.html), we learned how to represent colours as tuples of three intege (r, g, b), representing values for the amount of red, green, and blue used to form the colour. Some examples are shown in the table below.

RGB Value	Colour
(0, 0, 0)	
(255, 0, 0)	
(0, 255, 0)	
(0, 0, 255)	
(181, 57, 173)	
(255, 255, 255)	

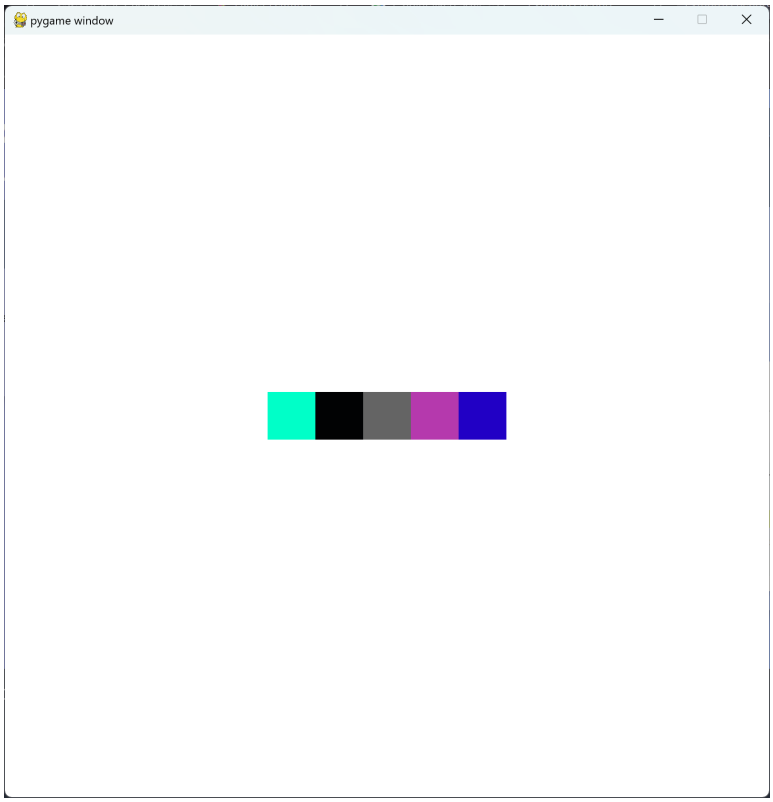
In Parts 4 and 5 of this assignment, you'll apply your knowledge of Python comprehensions to process collections of RGB colour tuples in interesting ways, and eventually build up a small Python program that allows you to manipulate real images!

First, we'll define a **colour row** (or **pixel row**) to be a list of RGB triples, represented in Python as a list of lists, where each inner list has exactly three elements. Here is an example of a colour row as a Python expression:

```
[[0, 255, 200], [1, 2, 3], [100, 100, 100], [181, 57, 173], [33, 0, 197]]
```

As the name "pixels" suggests, we'll see later how an image is composed of multiple colour rows, but in this part we'll keep things simple and write code that operates on just a single colour row.

0. **Warmup**. Open the `a1_part4.py` starter file. At the top of the file we've provided a function `warmup_part4` and instructions on how to complete it. Follow the instructions and then run the file in the Python Console, and call `warmup_part4()`. You should see a Pygame window appear with the colours you specified!



Tip: as you complete the functions in this part of the assignment, you'll want to check your work! You can use the doctest examples we provided, your own examples in the Python console, and/or call the provided `a1_helpers.show_colours_pygame` function to visualize your colour rows. Enjoy!

1. **Cropping rows**. Now, complete the functions `crop_row`, `crop_row_border_single`, and `crop_row_border_multiple`. Each of these functions will require you to take a colour row and return just part of that row.

   While this is possible to do using list slicing (a Python feature we'll discuss later in this course), fo this assignment you **must implement these functions using comprehensions (and without list slicing)**, as this is what we're evaluating on this assignment!

Hint: recall Exercise 2 of the Lecture 3 Worksheet (https://www.teach.cs.toronto.edu/~csc110y/fall/lectures/03-comprehensions-and-built-in-functions/worksheet/).

2. **Changing colours (reds)**. Next, let's look at how we can transform colour rows to create new rows. Implement the functions `remove_red_in_row` and `fade_red_in_row`, again using comprehensions.

   Hint: if you have an RGB tuple `colour`, you can access its individual red, green, and blue values l the indexing expressions `colour[0]`, `colour[1]`, and `colour[2]`, respectively.

3. **Changing colours (fade and blur)**. Finally, implement the functions `fade_row` and `blur_row`. These are the two most challenging functions on this assignment, and you'll need to combine the techniques you used in the previous two questions to complete them. We've given yo some hints in the function docstrings, so please read them carefully.

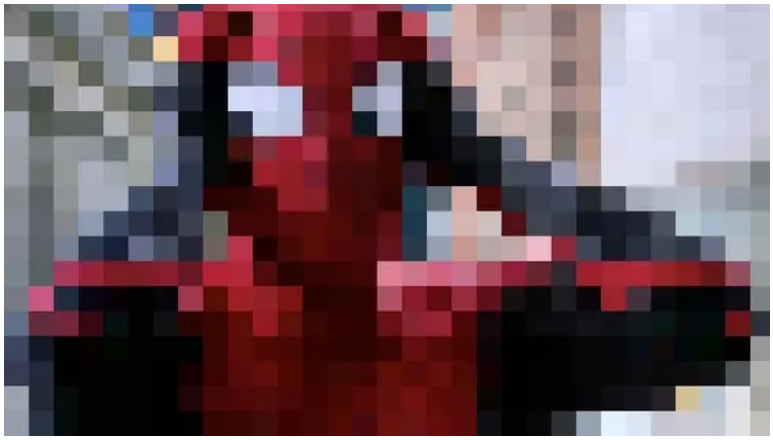
   Tip: these functions are also similar to the `gradient` function from Tutorial 1.

Part 5: Working with Image Data

In the previous part, you completed a series of functions that transform a row of colours. While we hop you had fun visualizing your results using Pygame, this may have felt a little unsatisfying: coloured squares is one thing, but what about real images? In this final part of Assignment 1, you'll apply the functions you wrote in Part 4 to work on real image data!

What is image data?

Every image we see on a computer screen is made up of small squares of various colours called *pixels*. More precisely, every image is a rectangular grid of pixels; typical image dimensions (e.g., "256-by-256" or "1920-by-1080") describe the number of columns and rows of pixels in the image—i.e., the image's width and height.



This is the key insight we'll use to extend our work from Part 4 to two-dimensional images: because images can be decomposed into rows of pixels, we can use comprehensions to apply our "colour row" functions to every row in an image!

0. **Warmup**. Open the `a1_part5.py` starter file. At the top of the file we've provided a function `warmup_part5`, which will return the pixel data for a small 30-by-15 image file we've provided under `images/spiderman.png`.

   a. First, try running this file in the Python console and then calling `warmup_part5()`. You should see a very long list of colour tuples! In fact, $30 \times 15 = 450$ tuples are displayed in total.

   ```
   >>> warmup_part5()
   [[125, 125, 125], [81, 87, 78], [103, 114, 117], [144, 151, 164], ...
   ```

   b. Let's use the Python library `pprint` to make the output look a bit nicer. In the Python console, type in the following:

   ```
   >>> image_data = warmup_part5()
   >>> import pprint
   >>> pprint.pprint(image_data)
   [[[125, 125, 125],
     [81, 87, 78],
     [103, 114, 117],
     [144, 151, 164],
     ...
   ```

Better! Notice that the double `[[` at the start. This confirms that the image data is a *list of lists*; if you scroll down, you'll see the "breaks" where one of the colour rows ends and anoth begins.

We can check the total number of colour rows contained in `image_data` using the `len` function:

```
>>> len(image_data)
15
```

This number corresponds to the *height* of the image.

c. Because `image_data` is a list, we can use list indexing to access the individual colour rows i contains. Try typing the following into the Python console to access the topmost colour row the image:

```
>>> image_data[0]
[[125, 125, 125], [81, 87, 78], [103, 114, 117], [144, 151, 164], ...
```

And because a colour row is itself a list, we can also find its size, which is the number of pixe in a single row of the image—i.e., the image's width.

```
>>> len(image_data[0])
30
```

d. Finally, inside the body of `warmup_part5`, try uncommenting (i.e., deleting the `#` and space the `a1_helpers.show_colour_rows.pygame` function call, and then re-running the file and calling `warmup_part5` again. You should see a new Pygame window appear showing th individual colours of the image! If you have the patience and time, you can count to make sure there are indeed 15 rows displayed, each with 30 squares.

Note: the Python interpreter will wait for you to close the Pygame window before continuing to the return statement of the function. This means you won't be able to see/access the colou rows in the Python console until you close the Pygame window!

1. **Transforming multiple colour rows.** Now that you've explored the data types we use to represent image data, let's actually get to transforming them! Your task is to complete the five functions `remove_red_in_image`, `fade_red_in_image`, `fade_rows_in_image`, `blur_rows_in_image`, and `crop_rows_in_image`.

This may seem like a lot, but you've already done the hard work in Part 4! Each of the functions you need to implement now should follow the same pattern: use a list comprehension together with one of your functions from Part 4 to apply a transformation to *each* colour row in image data

Tip: as in Part 4, you can check your work by writing doctests, using the Python console, and/or using the provided Pygame visualization function. We have provided one constant `EXAMPLE_PIXEL_GRID` that you might find helpful to use in your own tests, as you should be abl to calculate the exact expected values of every pixel for this small example.

> **Correction (Sept 16)**: in our original provided starter file, `EXAMPLE_PIXEL_GRID` represented colours as `tuple`s rather than `list`s. Please see the Assignment 1 FAQ (https://q.utoronto.ca/courses/278340/pages/assignment-1-faq-+-corrections?module_item_id=4047792) page for details.

2. **Generalized cropping.** Finally, implement the function `crop_image`, which enables cropping out both columns and rows. This last function follows a similar structure to the functions from Question 1, but you'll need to do something a bit different to select only a subset of the rows to keep.

   Once again, you may NOT use list slicing for this function!

Celebrate

Finally, let's see how all of your work can be used on some real images! Near the bottom of `a1_part5.py`, we've provided the skeleton of a function that you can use to read in an image file, perform some transformations, and then save the file.

> **Correction (Sept 16)**: in our original provided `a1_helpers.py`, the `save_image` function expected colours to be represented as tuples, not lists. Please see the Assignment 1 FAQ (https://q.utoronto.ca/courses/278340/pages/assignment-1-faq-+-corrections?module_item_id=4047792) page for details.

Experiment with this function on the provided files `images/spiderman.png` and `images/horses.png`, or copy your own images into the `images/` folder and use them instead! This last part is not for credit (we aren't grading any changes you make to the `transform_image` function,

except checking the code with PythonTA), so feel free to spend as much or as little time as you like on i But if someone asks you what you're working on for your first assignment in CSC110, you can use this function to show off images you've made! 😎

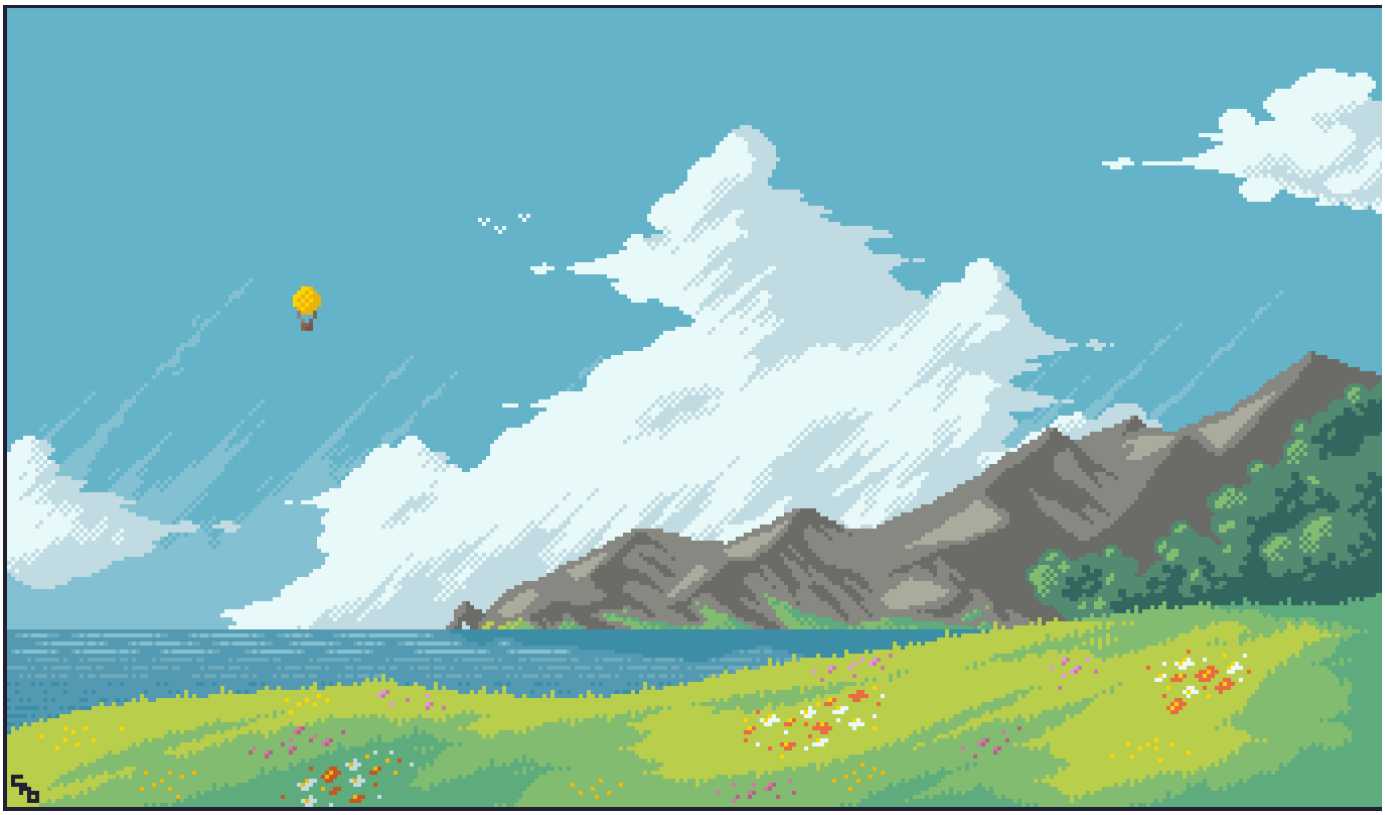
Submission instructions

Please **proofread** and **test** your work carefully before your final submission! As we explain in Runnin and Testing Your Code Before Submission (https://q.utoronto.ca/courses/278340/pages/running-and testing-your-code-before-submission?module_item_id=3953751), it is essential that your submitted code not contain syntax errors. **Python files that contain syntax errors will receive a grade of on all automated testing components (though they may receive partial or full credit on any TA grading for assignments).** You have lots of time to work on this assignment and check you work (and right-click -> "Run in Python Console"), so please make sure to do this regularly and fix syntax errors right away.

1. Login to MarkUs (https://markus.teach.cs.toronto.edu/2022-09) and go to the CSC110 course.
2. Go to Assignment 1, and the "Submissions" tab.
3. Submit the following files: `a1.tex`, `a1.pdf` (which must be generated from your `a1.tex` file), `a1_part2.py`, `a1_part3.py`, `a1_part4.py`, and `a1_part5.py`. Please note that MarkUs is picky with filenames, and so your filenames must match these exactly, including using lowercase letters.
    - Please check out the Overleaf tutorial on exporting project files (https://www.overleaf.com/learn/how-to/Exporting_your_work_from_Overleaf) to learn how to download your `a1.tex` and `a1.pdf` files from Overleaf.
4. Refresh the page, and then *download each file* to make sure you submitted the right version.

Remember, you can submit your files multiple times before the due date. So you can aim to submit you work early, and if you find an error or a place to improve before the due date, you can still make your changes and resubmit your work.

After you've submitted your work, please give yourself a well-deserved pat on the back and go take a re or do something fun or look at some art or eat some chocolate!



Serene Hillside by [On-3 (https://www.pixilart.com/on-3)](https://www.pixilart.com/on-3).